Action Concertée Incitative

SÉCURITÉ & INFORMATIQUE

APRON: Analyse de PROgrammes Numériques

Compte-rendu de la réunion du 28 juin 2005: Interface commune

Introduction

Ce document réorganise et synthétise les compte-rendus des réunions du 16 décembre 2004, 1er février 2005, 15 mars 2005 et 28 juin 2005.

Table des matières

Chapitre 1

Choix généraux

1.1 Sémantique d'une valeur abstraite

1. **Concrétisation d'une valeur abstraite.** Une valeur abstraite fournie par l'interface a pour concrétisation un sous-ensemble $X \subseteq \mathbb{N}^p \times \mathbb{R}^q$. Les variables sont donc typées, soit entières soit réelles.

2. . Les problèmes de précision et donc d'arithmétique modulaire pour les entiers 8, 16, 32 et 64 bits sont laissés à un niveau strictement supérieur à 1, voir dans l'analyseur lui-même au niveau de la sémantique concrète.

1.2 Niveaux d'interface

3. **Motivations.** L'objectif n'est pas d'avoir une liste d'opérateurs minimales, mais une liste d'opérateurs comportant les combinaisons d'opérateurs de base pouvant être simplifiées ou fréquemment utilisées par un analyseur.

Il s'agit d'assurer à la fois la performance des implantations et le confort de l'utilisateur, tout en évitant la duplication de code entre les différents librairies.

On propose donc de ne traiter au niveau 0 que les problèmes de performance (avantage algorithmique fort) et de reporter au niveau 1 les problèmes de confort, sachant que les problèmes de performance ne sont pas génériques, mais dépendent des domaines abstraits.

4. **Principe.** Il a été décidé d'établir différents niveaux d'interface.
   - Le niveau 0 est en prise directe avec la librairie sous-jacente (octogones, polyèdres) et suit les principes suivants :
     - On y place toutes les fonctions dont l'implémentation est spécifique au domaine abstrait et qui ne peuvent donc pas être partagées entre les librairies et .
     - L'interface est minimale : on en exclut les fonctions qui peuvent s'implémenter.à partir d'autres fonctions de niveau 0, **à moins qu'il n'y ait un avantage algorithmique fort** pour le laisser au niveau 0.
   - On réserve aux niveaux supérieurs les fonctions factorisables pour tous les domaines abstraits envisagés. Deux exemples envisagés :

1. L'appel automatique des opérations de redimensionnement et de permutations nécessaires pour calculer l'intersection $P1(x, y)$ avec $P2(y, z)$: ceci ne dépend pas du domaine abstrait considéré ; au pire, dans le cas de représentation interne creuses, il n'y a rien à faire.
2. L'abstraction d'expressions non-linéaires par des expressions linéaires d'intervalle.

On peut utiliser deux bibliothèques de niveau 0 pour les combiner de manière heuristique et obtenir une nouvelle bibliothèque de niveau 0.

Ci-dessous, points s'y rapportant :

Peut-on mettre au niveau 0 des fonctions spécifiques au domaine abstrait, par exemple une pré-analyse ?

Les produits cartésions pourraient être traités au niveau 1.

1.3 Langage de programmation

5. **Interface C** On définira au moins une version C de l'interface, qui servira de référence.

Le langage C a été choisi comme s'interfaçant aisément avec la plupart des langages (C++, Java, OCaml, Prolog, ...). C++ est moins adapté de ce point de vue. Par ailleurs, la plupart des librairies existantes sont en C (NewPolka, C3 (!?), octagones).

6. **Interface OCaml** L'IRISA et l'ENS sont intéressé par la définition d'une version OCaml.

Dans la suite, on ne discute que de l'interface C.

1.4 Formes normales des valeurs abstraites

La notion de forme(s) normale(s) a fait l'objet de vives discussions. Il s'agit de contrôler (abstraitement) la représentation interne des valeurs abstraites. Les besoins suivants ont été dégagés :

7. **Représentation minimale en terme d'occupation mémoire.** Cette notion existe dans les octogones (contraintes redondantes supprimées). Dans NewPolka cette notion n'est pas réellement fixée ; quelques possibilités :

1. Ensemble de contraintes non redondantes ;
2. Ensemble de contraintes *ou* de générateurs non-redondants, selon laquelle occupe le moins d'espace en mémoire ; c'est la solution correspondant à la définition ;
3. double représentation + matrice de saturation : c'est la seule notion de forme "normale" actuellement implantée dans NewPolka.

Dans les deux cas (octogones ou NewPolka), la forme minimale peut ne pas être adaptée aux calculs. Ainsi pour les octogones, la première chose à faire avant la plupart des opérations est de calculer la clôture de la forme minimale. Il en serait de même (dans une moindre mesure) pour la solution 2 dans NewPolka.

8. **Représentation canonique.** Il s'agit ici d'avoir une forme normale telle que deux valeurs sémantiquement égales ont exactement la même représentation.

La forme minimale précédente ne fournit pas forcément une représentation canonique. Par exemple, pour les polyèdres, il faut en outre normaliser la représentation de l'espace des égalités (ou des droites) et trier les contraintes (ou générateurs).

Problèmes se posent :
- Que faire lorsque les entiers ne sont que partiellement pris en compte ?
- Plus généralement, que faire lorsqu'on ne sait pas (ou on ne veut pas, pour des raisons de coût) la calculer ?

9. **Notion de réduction/approximation.** Dans certains cas, on veut contrôler assez finement la représentation interne, car elle a un impact sur la précision. Il s'agirait ici soit de simplifier la représentation, au prix éventuel d'une perte d'information (ex : on enlève des contraintes trop tarabiscotées, ou avec des coefficients de magnitude trop importante), soit d'affiner la représentation (ex de François pour la prise en compte des entiers, réduction dans le cas d'un domaine abstrait produit réduit de deux domaines abstraits) pour améliorer la précision d'opérations futures.

Une telle fonction sera fournie, paramétrée par un numéro d'algorithme. En outre, chaque opération sera paramétrée par la réduction/approximation à effectuer en entrée et en sortie de l'opération. Exemple(s) :
- Dans NEW POLKA existent des versions strictes et paresseuses des opérations. La version stricte travaille toujours sur des (doubles) représentations minimisées, tandis que les versions paresseuses ne recourt à Chernikova que si indispensable. Le paramétrage de chaque opération permettra de maintenir ce type de réglale, sans polluer l'API par d'innombrables versions de la même opération sémantique.

1.5 Fonctionnalités et architecture générale de l'interface

10. **Compatibilité avec les threads.** L'interface permettra d'écrire des implantations compatibles avec les threads.

11. **Gestionnaire.** Un contexte d'appel, objet de type `manager_t*`, sera explicitement passé à chaque fonction afin d'assurer les point suivants.

La transmission de données globales spécifiques à chaque librairie (mémoire de travail, options non fournies via l'interface commune, ...), ce qui assure en particulier la compatibilité avec les threads.

La transmission des options (leviers de réglages des algorithmes,etc ...). En effet, pour certains opérateurs ou même pour chaque opérateur abstrait, il est possible de sélectionner une implémentation particulière, l'implémentation par défaut, l'implémentation donnant le résultat le plus précis ou l'implémentation donnant un résultat le plus rapidement possible. La perte de généricité correspondante semble moins gênante que l'absence de cette flexibilité, au moins pour certains opérateurs particulièrement sensibles. Les trois implémentations de base peuvent être identiques : l'association numéro vers algorithme peut être surjective.

la gestion des exceptions (récupération des exceptions) ; les débordements en capacité (entiers), en temps et en espace sont détectés, ou tout au moins, les mécanismes nécessaires à la récupération des incidents sont définis. Le retour d'un résultat mathématiquement exact peut être identifié à la demande de l'utilisateur (projection entière, union vs enveloppe convexe, test de satisfiabilité en entier,...).

Le contexte n'est pas intégré aux objets du domaine abstrait pour éviter d'avoir à faire une vérification de compatibilité des contextes.

C n'offrant pas de mécanisme d'exceptions (sauf via le compliqué `setjmp`), et celui-ci étant spécifiques à chaque langage (C++, OCaml, Java), les informations d'exception sont retournées uniquement via la structure `manager_t`.

Le contenu du contexte d'appel n'est pas visible directement : c'est un objet opaque, fermé, avec des méthodes. Des primitives de construction et d'observation seront définies et fournies. [bj] Ce choix présente aussi l'avantage de faciliter l'interfaçage avec un langage comme OCaml.

12. **Implantations partielles.** Les implantations partielles sont acceptées, mais elles doivent offrir toutes les signatures prévues pour permettre l'édition de lien et l'échec éventuellement en cas d'appel à une fonction non implantée.

Lancement d'une exception `not_implemented` lorsqu'une fonction n'est pas implantée, et retour d'une valeur non spécifiée, le cas échéant (pointeur nul pour l'interface fonctionnelle ?)

13. **Gestion mémoire.** Les fonctions doivent être clairement documentées sur la façon dont elles gèrent la mémoire. On n'implante pas de mécanisme automatique de ramasse-miettes (compteur de référence ou autre) pour l'interface C.

L'interface OCaml en revanche utilisera les mécanismes du runtime OCaml (idem pour Prolog ou autre).

14. **Signatures fonctionnelles et impératives.** Les signatures fonctionnelles et impératives (ou destructives) sont toutes les deux supportées.

Questions résolues :

1. La représentation interne peut être changée sans préavis si la sémantique est conservée.
2. En mode destructif, la première valeur abstraite de la fonction est détruite par la fonction, l'utilisateur n'a plus à la gérer et ne doit pas l'utiliser. Seul le résultat (et les autres arguments) peuvent l'être.
3. Le mode destructif permet d'implémenter un mode effet de bord comme suit : `a = meet(man,destructive=true,a,b) ;` : on peut considérer que l'on fait un effet de brord sur `a`.
4. Pour les chaînes de caractères, on suit les conventions de C `string.h`.

15. **Opérations n-aires.** Les opérations n-aires (ex : borne supérieure) sont supportées en utilisant des tableaux d'arguments. Elles ne sont offertes qu'en version fonctionnelle.

La possibilité de passer plus de deux arguments n'est pas un simple confort. Elle a aussi un impact sur la fonctionalité dans le cas où on souhaite faire une union et où on a donc besoin de savoir si l'enveloppe convexe lui est égale. Idem pour les suites de projections. Enfin, elle a un impact sur la performance.

Le système `varargs` ne sera supporté que si un des participants en a le beosin (aucun pour l'instant). [bj] La version `vararg` est implantable en utilisant la version tableau, donc cela relève plutôt du niveau 1.

La définition d'une structure de liste propre à cette interface est exclue.

16. **Types des objets et mode de passage des paramètres.** Quelques rappels sur les idiomes C :

1. choix entre passage par valeur et par référence (pointeur) :

   ⟨*bidon*⟩≡
   ```
   void toto(object_t o);  /* appel par valeur sur le type object_t */
   void toto(object_t* o); /* appel par référence sur le type object_t */
   ```

   Avantage de l'appel par référence : plus rapide si `sizeof(object_t)` est sensiblement plus grand que la taille d'un scalaire (4 ou 8 octets). Inconvénient : casse-bonbon pour le programmeur de devoir écrire `toto(&x)` au lieu de `toto(x)`.

2. choix similaire pour la valeur retournée :

   ⟨*bidon*⟩+≡
   ```
   object_t toto();  /* retour par valeur */
   object_t* toto(); /* valeur de retour allouée par toto et à libérer par
                        l'appelant si nécessaire */
   ```

   Là, le retour par référence nécessite une allocation, donc plutôt moins bien point de vue efficacité !

3. type abstrait en C = type pointeur.

La dernière remarque justifie la signature suivante dans l'interface (*c.f.* §10.4), *si l'on choisit un passage par valeur* :

⟨*bidon*⟩+≡
```
abstract_t* abstract_meet(manager_t* man, abstract_t* a1, abstract_t* a2);
```

En effet tous les types impliqués sont considérés comme abstraits. Ici donc, on a un passage par valeur sur des types pointeurs, *et non un passage par référence sur les types* `manager_t` *et* `abstract_t`. C'est d'une certaine manière une question d'interprétation, bien sûr, mais en pratique si tous les objets sont déclarés, construits, manipulés avec un type `object_t*`, il n'y a jamais de déréférencement à faire et le programmeur n'a aucune question à se poser (c'est ce qui se passe dans la librairie de BDDs CUDD).

Maintenant, si le type `lincons_t` des contraintes linéaires utilisé dans `abstract0.h` est abstrait (généré à partir d'un type utilisateur, *c.f.* discussion dans §**??**), les signatures suivantes sont "imposées" (si passage par valeur) :

⟨*bidon*⟩+≡
```
abstract_t* abstract_meet_lincons(manager_t* man,
                                  abstract_t* a, lincons_t* c);
abstract_t* abstract_meet_lincons_array(manager_t* man,
                                        abstract_t* a,
                                        lincons_t** tlincons, int size);
```

Une contrainte linéaire étant de type `lincons_t*`, les tableaux de contraintes linéaires sont de type `lincons_t**`.

Mais si le type `lincons_t` utilisés dans ces fonctions est utilisateur (public), on pourrait *aussi* utiliser (toujours si passage par valeur)

⟨*bidon*⟩+≡
```
abstract_t* abstract_meet_lincons(manager_t* man,
                                  abstract_t* a, lincons_t c);
abstract_t* abstract_meet_lincons_array(manager_t* man,
                                        abstract_t* a,
                                        lincons_t* tlincons, int size);
```

Dans l'interface proposée, les conventions suivies sont :
- Appel et retour par valeur.
- Les types des objets (`abstract_t*, linexpr_t*, ray_t*`) sont en général des types pointeurs, sauf pour :
  - Les nombres (`coeff_t`) ;
  - Les intervalles (`interval_t`),
  - Les contraintes (`lincons_t`)

  Ces trois derniers types sont considérés comme des types utilisateurs *c.f.* §2, et leur taille (en terme de `sizeof()`) reste petite (mais ces structures peuvent bien sûr contenir des pointeurs vers des zones de taille importante, *e.g.*. nombres multi-précision).
- On utilisera les types `size_t` (défini dans `stddef.h`) et `dim_t` (défini par `unsigned int`) pour typer les tailles (de tableaux, par exemple) et les dimensions (demande/suggestion de l'ENS)

17. **Retour d'arguments multiples.** Quelle politique lorsque plusieurs arguments doivent être retournés ? Par exemple, si une fonction retourne un tableau d'objets de taille non connue à l'avance, il faut retourner un pointeur sur un tableau plus la taille du tableau.

Essaye-t-on d'unifier l'interface ? En C, lorsque `toto(x)` doit retourner deux arguments `a` et `b`, on a 3 solutions ;

⟨*bidon*⟩+≡
```
toto(x,&a,&b); /* paramètres résultats, pasq très joli; */

a=toto(x,&b);  /* une valeur retournée, l'autre en paramètre résultat;
                  guère plus joli, et très moche si a et b ont une
                  signification "symmétriques"
                */

struct titi { a_t a; b_t b; };
titi=toto(x); /* creation d'un nouveau type, parfois
                 juste pour une seule fonction */Ah, ces \verb|*%$č&#| de C, C++, Java!
```

Au cours de la réunion de fin juin 2005, on a opté pour la dernière solution.

Pour l'interface OCaml, qu'on essayera de générer un peu automatiquement à partir de l'interface C, à noter que CamlIDL sait assez bien traiter les paramètres résultats, en les transformant en valeurs retournées dans un n-uplet.

18. **Représentation interne des nombres.** Le type numérique utilisé en interne dans une librairie sous-jacente est défini à la compilation, e.g. via une option `-D`, et/ou à l'édition de lien (comme dans PIPS, type *Value, et dans Polka, type* pkint).

Un type numérique utilisateur est fourni par l'interface, *c.f.* §2.

19. **Nommage des fonctions.** On utilisera des préfixes pour distinguer les différentes bibliothèques en C ? Par exemple, OCT pour octagone, PPL pour Parma, HQ proposé par Duong. Préfixes envisagés : POLKA, C3, PIPS, POLYLIB.

Doit-on ensuite utiliser des macros pour rendre génériques ? Ou utiliser une technique objet (cf ci-dessous).

[bj] Technique objet : on peut inclure dans la structure `manager_t` des pointeurs sur les fonctions offertes par l'interface, et définir des fonctions génériques qui utilisent `manager_t` pour appeler la fonction effective. C'est ce qui me semble le plus souple, car en changeant de manager on change appelée.

⟨*bidon*⟩+≡
```
abstract_t* meet(manager_t* man, abstract_t* a1, abstract_t* a2)
{
  return manager->abstract_meet(manager,a1,a2);
}
```

20. **Sérialisation/désérialisation.** La sérialisation/désérialisation des objets (valeurs abstraites, contraintes, ...) sera fournie par l'interface.

Si la sérialisation se fait sur les types utilisateurs (contraintes notamment), alors possibilité d'échanger entre plusieurs librairies. Si elle se fait sur un type abstrait, ce n'est plus le cas (sans compter le problème de la représentation des coefficients dans les types internes, dépendant d'options de compilation de la même librairie).

21. **Conversions et super-treillis.** Afin de faciliter les conversions entre valeurs abstraites de domaines différents, on peut considérer un super-treillis et n'implanter que les conversions d'un domaine abstrait vers ce super-treillis.

En l'absence d'informations de congruence, les polyèdres convexes suffisent pour tous les treillis abstraits envisagés (intervalles, égalités, octogones, templtate constraints, octaèdres, polyèdres convexes). Toutefois, si on veut ajouter les égalités polynomiales de degré borné ([Seidl]), ce n'est plus vrai.

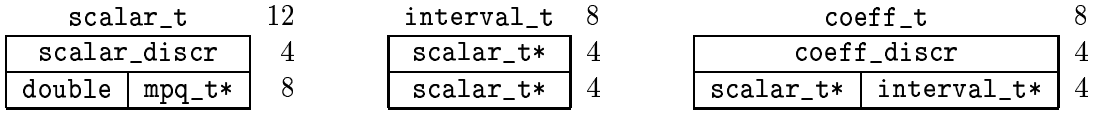
Si on rajoute les congruences, il faut plus. Presburger est un peu candidat, mais quid des réels ?

Au sujet d'un super-treillis, remarque anonyme : une syntaxe peut suffire.

Chapitre 2

Représentation des coefficients, fichiers `ap_scalar.h`, `ap_interval.h` et `ap_coeff.h`

Voici un résumé graphique des différents types, avec leur taille en octets sur une machine 32 bits.



Tous ces types de données sont manipulés via des pointeurs, avec des fonctions d'initialisation et de nettoyage `X_t* X_alloc()` et `void X_free(X_t*)`.

22. **Représentation des nombres dans les librairies sous-jacentes et types abstraits.** Elle est entièrement libre.
- Les entiers peuvent être représenté par des `int`, `long int`, `long long int`, ou des entiers multi-précisions (d'une librairie quelconque).
- les réels peuvent être représentés par des rationnels sur les entiers précédents, ou par des flottants (`float`, `double` ou flottants multi-précisions), ou même des intervalles de flottants (!? pas sûr, attends confirmation de l'ENS ; sans doute nécessaire pour les conversions).

23. **Scalaires et leurs représentations dans les types utilisateurs.** Il est important de distinguer le domaine concret des scalaires et leur représentation dans les expressions, contraintes, générateurs, ...

Les dimensions étant soit entières, soit réelles, on peut considérer qu'il en est de même pour les scalaires. En concret, les entiers étant un sous-ensemble des réels, on peut se limiter aux réels. Mais cette propriété d'inclusion n'est pas vérifiée au niveau de la représentation (voir exemple précédent). Typiquement, un entier multi-précision n'est pas toujours représentable de manière exacte par un `float`.

2.1 Représentation des nombres

24. **Proposition de BJ**

⟨*bidon*⟩≡

```
typedef enum ap_scalar_discr_t {
  AP_SCALAR_DOUBLE, /* flottant avec double */
  AP_SCALAR_MPQ,   /* rationnel avec multiprécision de GMP */
} ap_scalar_discr_t;

typedef struct ap_scalar_t {
  ap_scalar_discr_t discr;
  union {
    double dbl;
    mpq_ptr mpq; /* +infty coded by 1/0, -infty coded by -1/0 */
  } val;
} ap_scalar_t;

/* follows all the necessary operations on numbers of type ap_scalar_t */
```

Commentaires :

- Les rationnels multi-précisions incluent le cas des rationnels sur des entiers à précision fixe (`int`, `long int`, etc).
- `double` inclut `float`. Toutefois, pour plus de généralité, on pourrait même prendre des flottants multi-précisions. Antoine Miné autorise `mpfr` dans ses octogones, mais cette version est-elle réellement utilisée ?
- On autorise ici que chaque scalaire apparaissant dans une expression puisse avoir un type différent (rationnel ou flottant). On pourrait être moins flexible en imposant dans chaque expression un type unique pour les scalaires (le champ de type `ap_scalar_discr_t` serait alors associé à l'expression). Avantage : lors de l'addition de deux expressions "2x+3y+4z" et "x+y", pas besoin de vérifier à tout bout de champ la compatibilité des arguments.

[bj] Je pense qu'il s'agit d'un point important pour la prochaine réunion. Antoine a plus d'expérience que moi avec sa librairie d'octogones, et l'ENS en général pour l'utilisation des flottants/intervalles de flottants

2.2 Intervalles

25. **Intervalles non bornés** (pour les expressions linéaires d'intervalles, pour les extracteurs de propriétés)

⟨*bidon*⟩+≡

```
typedef struct ap_interval_t {
  ap_scalar_t* inf;
  ap_scalar_t* sup;
} ap_interval_t;
```

2.3 Coefficients

Utilisées dans les expressions linéaires et linéaires d'intervalle.

⟨*bidon*⟩+≡

```
typedef enum ap_coeff_discr_t {
  AP_COEFF_SCALAR,
  AP_COEFF_INTERVAL
} ap_coeff_discr_t;
  /* Discriminant for coefficients */

typedef struct ap_coeff_t {
  ap_coeff_discr_t discr; /* discriminant for coefficient */
  union {
    ap_scalar_t* scalar;         /* cst (normal linear expression) */
    ap_interval_t* interval;   /* interval (quasi-linear expression) */
  } val;
} ap_coeff_t;
```

Chapitre 3

Dimension : dimensions et opérations reliées, fichier `ap_dimension.h`

Ce fichier définit les types de bases pour manipuler les dimensions.

⟨*⟩≡

```
/* ********************************************************************** */
/* ap_dimension.h : dimensions and related operations */
/* ********************************************************************** */

/* This file is part of the APRON Library, released under LGPL license.  Please
   read the COPYING file packaged in the distribution */

/* GENERATED FROM environment.nw : DO NOT MODIFY! */

#ifndef _AP_DIMENSION_H_
#define _AP_DIMENSION_H_

#include <stdlib.h>
#include <stdio.h>

#ifdef __cplusplus
extern "C" {
#endif
```

3.1 Datatypes

⟨*⟩+≡

```
/* ========================================================================== */
/* Datatypes */
/* ========================================================================== */

/* Datatype for dimensions */
typedef unsigned int ap_dim_t;
#define AP_DIM_MAX UINT_MAX
/* Used for sparse representations (mean : to be ignored) and also as
   a result when an error occured */

/* Datatype for specifying the dimensionality of an abstract value */
typedef struct ap_dimension_t {
  size_t intdim;
  size_t realdim;
} ap_dimension_t;

/* Datatype for specifying change of dimension */
typedef struct ap_dimchange_t {
  ap_dim_t* dim;     /* Assumed to be an array of size intdim+realdim */
  size_t intdim; /* Number of integer dimensions to add/remove */
  size_t realdim; /* Number of real dimensions to add/remove */
} ap_dimchange_t;

/* The semantics is the following :

- Addition of dimensions :

  dimchange.dim[k] means : add one dimension at dimension k and shift the
  already existing dimensions greater than or equal to k one step on the right
  (or increment them).

  if k is equal to the size of the vector, then it means : add a dimension at
  the end.

  Repetion are allowed, and means that one inserts more than one dimensions.

  Example :
  linexpr0_add_dimensions([i0 i1 r0 r1], { [0 1 2 2 4],3,1 }) returns
  [0 i0 0 i1 0 0 r0 r1 0], considered as a vector with 5 integer dimensions
  and 4 real dimensions.

- Removal of dimensions

  dimchange.dim[k] means : remove the dimension k and shift the dimensions
  greater than k one step on the left (or decrement them).

  Repetitions are meaningless (and are not correct specification)

  Example :
  linexpr0_remove_dimensions([i0 i1 i2 r0 r1 r2], { [0 2 4],2,1 }) returns
  [i1 r0 r2], considered as a vector with 1 integer dimensions
  and 2 real dimensions.
```

```
*/

/* Datatype for permutations */
typedef struct ap_dimperm_t {
  ap_dim_t* dim;    /* Array assumed to be of size size */
  size_t size;
} ap_dimperm_t;
  /* Such an object represent the permutation
     i -> dimperm.p[i] for 0<=i<dimperm.size */
```

3.2 Functions

⟨*⟩+≡

```
/* ============================================================ */
/* Functions */
/* ============================================================ */

void ap_dimchange_init(ap_dimchange_t* dimchange, size_t intdim, size_t realdim);
  /* Initialize a dimchange structure (allocate internal array) */
ap_dimchange_t* ap_dimchange_alloc(size_t intdim, size_t realdim);
  /* Allocate and initialize a dimchange structure */

static inline
void ap_dimchange_clear(ap_dimchange_t* dimchange)
{ if (dimchange->dim) free(dimchange->dim); dimchange->intdim = dimchange->realdim = 0; dimchange->di
  /* Clear a dimchange structure (deallocate internal arrau) */
static inline
void ap_dimchange_free(ap_dimchange_t* dimchange)
{ ap_dimchange_clear(dimchange); free(dimchange); }
  /* Deallocate and clear a dimchange structure */

void ap_dimchange_fprint(FILE* stream, ap_dimchange_t* dimchange);
  /* Printing */
void ap_dimchange_add_invert(ap_dimchange_t* dimchange);
  /* Assuming that dimchange is a transformation for add_dimensions,
     invert it to obtain the inverse transformation using remove_dimensions */
```

⟨*⟩+≡

```
  void ap_dimperm_init(ap_dimperm_t* dimperm, size_t size);
    /* Initialize a dimperm structure (allocate internal array) */
  ap_dimperm_t* ap_dimperm_alloc(size_t size);
    /* Allocate and initialize a dimperm structure */

  static inline
  void ap_dimperm_clear(ap_dimperm_t* dimperm)
  { if (dimperm->dim) free(dimperm->dim); dimperm->size = 0; dimperm->dim = NULL; }
    /* Clear a dimperm structure (deallocate internal arrau) */
  static inline
  void ap_dimperm_free(ap_dimperm_t* dimperm)
  { ap_dimperm_clear(dimperm); free(dimperm); }
    /* Deallocate and clear a dimchange structure */

  void ap_dimperm_fprint(FILE* stream, ap_dimperm_t* perm);
    /* Print a permutation under the form :
       dimperm : size=...
       0 -> perm->dim[0]
       1 -> perm->dim[1]
       ...
   */

  void ap_dimperm_set_id(ap_dimperm_t* perm);
    /* Generate the identity permutation */

  void ap_dimperm_compose(ap_dimperm_t* perm, const ap_dimperm_t* perm1, const ap_dimperm_t* perm2);
    /* Compose the 2 permutations perm1 and perm2 (in this order)
       and store the result the already allocated perm.
       The sizes of permutations are supposed to be equal.
       At exit, we have perm.dim[i] = perm2.dim[perm1.dim[i]]
    */
  void ap_dimperm_invert(ap_dimperm_t* nperm, const ap_dimperm_t* perm);
    /* Invert the permutation perm and store it in the already allocated nperm.
       The sizes of permutations are supposed to be equal.
    */
```

⟨*⟩+≡

```
  #ifdef __cplusplus
  }
  #endif

  #endif
```

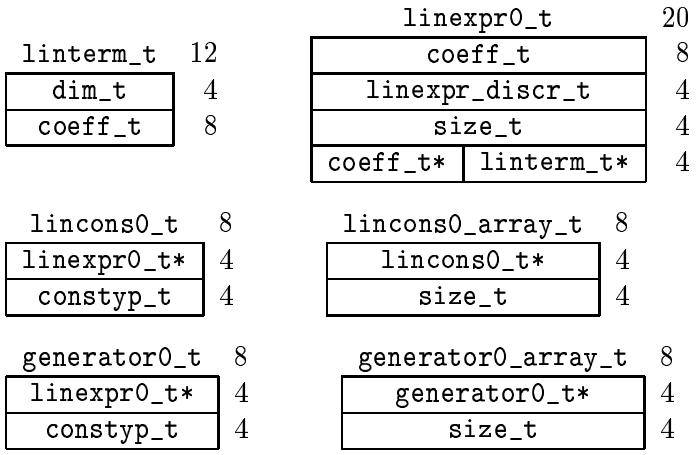
Chapitre 4

Sémantique concrète au niveau 0 et fichier ap_expr0.h

⟨*⟩≡

```
/* ********************************************************************** */
/* ap_expr0.h : linear expressions, constraints and generators */
/* ********************************************************************** */

/* This file is part of the APRON Library, released under LGPL license.  Please
   read the COPYING file packaged in the distribution */

/* GENERATED FROM ap_expr0.nw : DO NOT MODIFY! */

#ifndef _AP_EXPR0_H_
#define _AP_EXPR0_H_

#include "ap_linexpr0.h"
#include "ap_lincons0.h"
#include "ap_generator0.h"

#endif
```

Voici un résumé graphique des différents types, avec leur taille en octets sur une machine 32 bits :



`ap_linexpr0_t` est toujours déclaré comme un pointeur (`ap_linexpr0_t* expr`), et alloué ou libéré par `ap_linexpr0_t* ap_linexpr0_alloc(...)` et `ap_linexpr0_free(ap_linexpr0_t*)`. A noter, on a mis les coefficients en-ligne, pour éviter une indirection, ce qui viole la convention sur la manipulaiton des coefficients (normalement toujours manipulés via des pointeurs). Ce n'est pas un inconvénient, car `ap_linexpr0_t` est considéré comme un type abstrait, même par l'implémenteur d'une librairie.

Les autres types (`ap_lincons0_t`,`ap_ap_lincons0_array_t`, `ap_generator0_t` et `ap_ap_generator0_array_t`) sont manipulés via des pointeurs, mais déclarés "en-ligne" (`X_t obj ;`), avec des fonctions d'initialisation et de nettoyage `void X_init(X_t*)` et `void X_clear(X_t*)`.

4.1 Types de donnée

4.1.1 Dimensions

26. **Synthèse des décisions prises.**

1. Au niveau 0, la concrétisation d'une valeur abstraite est un sous-ensemble $X \subseteq \mathbb{N}^p \times \mathbb{R}^q$.
2. Les dimensions sont numérotées de 0 à $p+q-1$ et sont donc typées. Une dimension i est entière si $0 \leq i < p$, réelle sinon.
3. La dimension (p, q) de l'espace $\mathbb{N}^p \times \mathbb{R}^q$ dans lequel un sous-ensemble abstrait est plongé est accessible par un opérateur.
4. Des arguments sont compatibles au niveau 0 s'ils ont tous la même dimension.
5. Les opérations peuvent ignorer le caractère entier d'une dimension, s'il en résulte une surapproximation (exemple : ce que fait NewPolka).
6. La gestion de la liaison entre dimension et zone de mémoire abstraite (e.g. identificateurs dans les cas simples, adresses mémoires, ensemble d'adresses mémoire) est déléguée au niveau 1.

Pour les changements de dimensions, une première interface consistait à
- ajouter des dimensions à la fin puis permuter,
- ou permuter puis enlever les dimensions à la fin.

À la réunion du 9 février 2006 lui a été préférée l'interface suivante :

– insertion de dimensions n'importe où ;
– retrait de dimensions n'importe où ;
– permutations sans ajout ou retrait.

4.1.2 Expressions linéaires et extensions

27. **Duplication type utilisateur/type interne.** L'interface définit un type utilisateur d'expression, qui ne correspond en général pas au type interne utilisé par une librairie sous-jacente. Deux solutions sont possibles, que l'on illustre sur une fonction d'affectation du type `abstract_t* abstract_assign_linexpr0(abstract_t* a, dim_t dim, ap_linexpr0_t expr)`.

1. La fonction `abstract_assign_linexpr0` prend en entrée le type public des expressions `ap_linexpr0_t` et effectue la conversion vers le type interne à la volée ;
2. Il existe dans l'interface un type interne (opaque) explicite, avec les fonctions de conversions nécessaires (ou les constructeurs nécessaires), et `abstract_assign_linexpr0` prend en entrée le type interne.

Le choix 1 est plus pratique d'un point de vue utilisateur, mais les conversions peuvent être coûteuses, d'autant qu'elles incluent également des conversions de représentations des coefficients. Le choix 2 est plus économique, mais alourdit l'interface et son utilisaton.

Questions liées :

1. On a considéré le cas du passage d'objet à une fonction interne, mais il y a aussi le cas du retour à considérer. [bj] A mon avis, autant que ce soit homogène : si on passe des types internes, autant retourner des types internes, même si souvent (mais pas toujours) la valeur retournée va être examinée par l'appelant, ce qui nécessite une conversion vers le type utilisateur.
2. Si on fournit la sérialisation des expressions, faut-il la fournir à la fois pour le type utilisateur et le type interne ?

28. **Type utilisateur pour les expressions linéaires (d'intervalle).**

Les représentations denses et creuses ont toutes deux leur intérêt. On propose donc un type union avec un champ discriminant. Les expressions linéaires sont en fait des expressions linéaires d'intervalle, au sens de Miné, c'est-à-dire que les coefficients peuvent être des intervalles. [bj] Pour l'instant, on ne type pas les dimensions dans les expressions.

29. **Type utilisateur séparé pour les expressions linéaires d'intervalle**

Ces expressions permettent d'abstraire des expressions non-linéaires, et aussi de prendre en compte la sémantique spécifique des flottants.

La transformation d'expressions non-linéaires en expressions linéaires d'intervalle, puis en expressions quasi-linéaires (algorithme(s) proposés dans la thèse d'Antoine Miné) est indépendante du treillis abstraits utilisés, mais dans le cas des octogones, Antoine Miné préfère se brancher directement sur les expressions linéaires d'intervalle. Donc toutes les expressions linéaires sont des expressions linéaires d'intervalle, mais on offre une fonction générale de conversion d'une expression linéaire d'intervalle en expression quasi-linéaire utilisable par toute librairie sous-jacente.

Avoir des types séparés pour les expressions quasi-linéaires et linéaires d'intervalle alourdit sans doute trop l'interface (duplication des types contraintes, tableaux de contraintes, duplication

des fonctions, etc...). Mais du coup c'est à la librairie sous-jacente (à l'aide de fonctions fournies par l'interface) de se coltiner le décorticage des fonctions linéaires d'intervalles.

30. **Manipulation d'expressions.** Doit-on fournir des opérations de manipulation d'expressions ? [bj] Oui

4.1.3 Contraintes linéaires

4.1.4 Générateurs

31. Obtenir les générateurs d'une valeur abstraite est utile pour certaines applications (génération de code). Par ailleurs, la connaissance des rayons permet de savoir quelles sont les directions infinies dans une valeur abstraite. On utilise le type des expressions linéaires d'intervalle, mais avec l'hypothèse qu'elles se réduisent à des expressions linéaires scalaires.

4.1.5 Expressions et contraintes non linéaires

32. Le cas des contraintes non linéaires n'a pas été encore abordé. Une solution viable semble être d'abstraite, au niveau 1 de l'interface, les expressions non-linéaires par des expressions linéaires d'intervalle. Il n'est pas sûr que l'on puisse faire beaucoup mieux en laissant chaque domaine abstrait se débrouiller avec les expressions non linéaires (*i.e.*, le code de conversion vers les expressions linéaires d'intervalle risque d'être dupliqué).

4.1.6 Congruences

33. On ne dispose d'aucune implémentation des congruences, et donc d'aucun exemple sous la main. Le type `ap_generator0_t` permet en principe de représenter des informations de congruence sur les entiers.

Chapitre 5

Contexte d'appel et fichier `manager.h`

Ce chapitre définit le fichier `manager.h` se rapportant à la définition du contexte d'appel, ainsi que des types d'usage généraux.

⟨*⟩≡

```
/* ********************************************************************** */
/* manager.h : global manager passed to all functions */
/* ********************************************************************** */

/* This file is part of the APRON Library, released under LGPL license.  Please
   read the COPYING file packaged in the distribution */

#ifndef _AP_MANAGER_H_
#define _AP_MANAGER_H_

#include <stdlib.h>
#include <stdio.h>

#include "ap_coeff.h"

#ifdef __cplusplus
extern "C" {
#endif
```

5.1 Types d'usage général

⟨*⟩+≡

```
/* ********************************************************************** */
/* I. Types */
/* ********************************************************************** */

/* ====================================================================== */
/* I.0 General usage */
/* ====================================================================== */

/* Boolean with a third value */
typedef enum tbool_t {
  tbool_false=0,
  tbool_true=1,
  tbool_top=2,   /* don't know */
} tbool_t;

static inline tbool_t tbool_of_bool(bool a);
static inline tbool_t tbool_of_int(int n);
static inline tbool_t tbool_or(tbool_t a, tbool_t b);
static inline tbool_t tbool_and(tbool_t a, tbool_t b);

/* For serialization */
typedef struct ap_membuf_t {
  void* ptr;
  size_t size;
} ap_membuf_t;
```

5.2 Identification des fonctions

34. **Identification des fonctions.** Chaque fonction de l'interface niveau 0 est identifiée par un numéro, allant de `0` à `FUNID_SIZE-1`. Les motivations sont les suivantes :

Options associées aux fonctions : cette solution permet d'avoir un tableau d'options pour les options associées à chaque fonction ; cette solution réduit considérablement le nombre de fonctions permettant de consulter et de modifier ces options.

Appels virtuels : de même, on peut inclure dans le gestionnaire un tableau de fonctions, qui permet de se servir du gestionnaire pour faire des appels virtuels : c'est le gestionnaire qui définit la librairie effectivement appelée (voir plus loin).

Exceptions : on a également un identifiant "universel" des fonctions concernées dans les exceptions.

⟨*⟩+≡

```
/* ========================================================================== */
/* I.1 Identifying functions */
/* ========================================================================== */

typedef enum ap_funid_t {
  AP_FUNID_UNKNOWN,
  AP_FUNID_COPY,
  AP_FUNID_FREE,
  AP_FUNID_ASIZE, /* For avoiding name conflict with AP_FUNID_SIZE */
  AP_FUNID_MINIMIZE,
  AP_FUNID_CANONICALIZE,
  AP_FUNID_APPROXIMATE,
  AP_FUNID_IS_MINIMAL,
  AP_FUNID_IS_CANONICAL,
  AP_FUNID_FPRINT,
  AP_FUNID_FPRINTDIFF,
  AP_FUNID_FDUMP,
  AP_FUNID_SERIALIZE_RAW,
  AP_FUNID_DESERIALIZE_RAW,
  AP_FUNID_BOTTOM,
  AP_FUNID_TOP,
  AP_FUNID_OF_BOX,
  AP_FUNID_OF_LINCONS_ARRAY,
  AP_FUNID_DIMENSION,
  AP_FUNID_IS_BOTTOM,
  AP_FUNID_IS_TOP,
  AP_FUNID_IS_LEQ,
  AP_FUNID_IS_EQ,
  AP_FUNID_IS_DIMENSION_UNCONSTRAINED,
  AP_FUNID_SAT_INTERVAL,
  AP_FUNID_SAT_LINCONS,
  AP_FUNID_BOUND_DIMENSION,
  AP_FUNID_BOUND_LINEXPR,
  AP_FUNID_TO_BOX,
  AP_FUNID_TO_LINCONS_ARRAY,
  AP_FUNID_TO_GENERATOR_ARRAY,
  AP_FUNID_MEET,
  AP_FUNID_MEET_ARRAY,
```

```
  AP_FUNID_MEET_LINCONS_ARRAY,
  AP_FUNID_JOIN,
  AP_FUNID_JOIN_ARRAY,
  AP_FUNID_ADD_RAY_ARRAY,
  AP_FUNID_ASSIGN_LINEXPR,
  AP_FUNID_ASSIGN_LINEXPR_ARRAY,
  AP_FUNID_SUBSTITUTE_LINEXPR,
  AP_FUNID_SUBSTITUTE_LINEXPR_ARRAY,
  AP_FUNID_ADD_DIMENSIONS,
  AP_FUNID_REMOVE_DIMENSIONS,
  AP_FUNID_PERMUTE_DIMENSIONS,
  AP_FUNID_FORGET_ARRAY,
  AP_FUNID_EXPAND,
  AP_FUNID_FOLD,
  AP_FUNID_WIDENING,
  AP_FUNID_CLOSURE,
  AP_FUNID_SIZE,
  AP_FUNID_CHANGE_ENVIRONMENT,
  AP_FUNID_RENAME_ARRAY,
  AP_FUNID_SIZE2
} ap_funid_t;

extern const char* ap_name_of_funid[AP_FUNID_SIZE2];
/* give the name of a function identifier */
```

5.3 Exceptions

35. **Type** le type `exc_t` est défini comme suit :

⟨*⟩+≡

```
/* ================================================================== */
/* I.2 Exceptions */
/* ================================================================== */

/* Exceptions (public type) */
typedef enum ap_exc_t {
  AP_EXC_NONE,           /* no exception detected */
  AP_EXC_TIMEOUT,        /* timeout detected */
  AP_EXC_OUT_OF_SPACE,   /* out of space detected */
  AP_EXC_OVERFLOW,        /* magnitude overflow detected */
  AP_EXC_INVALID_ARGUMENT, /* invalid arguments */
  AP_EXC_NOT_IMPLEMENTED, /* not implemented */
  AP_EXC_SIZE
} ap_exc_t;

extern const char* ap_name_of_exception[AP_EXC_SIZE];

/* Exception log */
typedef struct ap_exclog_t {
  ap_exc_t exn;
  ap_funid_t funid;
  char* msg;                    /* dynamically allocated */
  struct ap_exclog_t* tail;
} ap_exclog_t;

/* Exceptions and other indications (out) (opaque type) */
typedef struct ap_result_t {
  ap_exclog_t* exclog; /* history of exceptions */
  ap_exc_t exn;                /* exception for the last called function */
  tbool_t flag_exact; /* result is mathematically exact or not
                         or don't know */
  tbool_t flag_best;  /* result is best correct approximation or not
                         or don't know */
} ap_result_t;
```

Remarque importante : dans la précédente proposition, on avait des flags d'exception (un par exception) et se posait une question non tranchée : suppose-t-on que les flags d'exceptions sont implicitement remis à faux au début de chaque appel de fonction, ou faut-il le faire manuellement ? Ici, l'utilisation d'un type énuméré résoud le choix en faveur de la première option.

36. **Discussion :** Le CRI souhaite que tout type utilisé comme résultat d'une fonction inclut une constante *xxx_undefined*. La discussion n'a pas été terminée lors de la première réunion sur ce point, l'étude de *is_bottom* ne s'y prêtant pas.

La valeur *undefined* n'apparait pas dans les domaines abstraits.

Discussion à reprendre :

- l'information ne devrait-elle pas être portée dans le *manager* ? Non pour le CRI qui se place dans une perspective de mise au point.
- *undefined* n'est-il pas *top* ? Le CRI doit effectivement utiliser parfois *undefined* comme *top* pour alléger l'implémentation. La sémantique d'*undefined* est-elle bien définie ? Top ? Not implemented ? Exception occured ? Utilisation des codes d'exception ?
- *fail_with* ? ! ?
- Jérôme : utilisation pour des effets de bord détectés mais non traités.
- propagation des erreurs ? des exceptions ? remises à zéro du *manger* ?

5.4 Options

On synthétise les décisions prises (ou en cours) avant de refléter les discussions à ce sujet.

37. **Options associées à chaque opération.** À chaque opération (non triviale) sera associée un objet du type suivant :

⟨*⟩+≡

```
/* ========================================================================== */
/* I.2 Options */
/* ========================================================================== */

/* Option associated to each function (public type) */
typedef struct ap_funopt_t {
  int algorithm;
  /* Algorithm selection :
     - 0 is default algorithm;
     - MAX_INT is most accurate available;
     - MIN_INT is most efficient available;
     - otherwise, no accuracy or speed meaning
  */
  int approx_before;
  int approx_after;
  /* Related to the notion of approximation/reduction.
     Indicates which kind of "approximation" may be performed on the
     argument(s) before the algorithm, and on the result delivered
     by the algorithm. 0 is default behaviour. */
  size_t timeout; /* unit!? */
  /* Above the given computation time, the function may abort with the
     exception flag flag_time_out on.
  */
  size_t max_object_size; /* in abstract object size unit. */
```

```
    /* If during the computation, the size of some object reach this limit, the
       function may abort with the exception flag flag_out_of_space on.
    */
    bool flag_exact_wanted;
    /* return information about exactitude if possible
    */
    bool flag_best_wanted;
    /* return information about best correct approximation if possible
    */
  } ap_funopt_t;
```

On a choisi de spécifier ce type d'options fonction par fonction, afin d'éviter à avoir à repositionner des options globales entre chaque appel.

Le retour d'un résultat mathématiquement exact peut être identifié à la demande de l'utilisateur (projection entière, union vs enveloppe convexe, test de satisfiabilité en entier,...).

Idem pour un résultat représentant la meilleure approximation correcte de l'opération dans le domaine abstrait considéré.

38. **Taille abstraite.** Une notion de taille abstraite d'objet est définie. Disposer d'une taille concrète (en octets) est envisageable mais potentiellement très coûteux à évaluer (notamment lors de l'emploi de nombres en multi-précision). or il est souhaitable que le mécanisme d'"out_of_space" ne pénalise pas trop les performances.

39. **Mécanisme de détection des *timeout*** : threads concurrentes ? Exception ? Discussion au sein de la Polylib pour avantages et inconvénients. [bj] Apparemment, on a opté pour exception.

40. **Options paramétrées.** Certaines heuristiques peuvent nécessiter des paramètres. Comment les passe-t-on ? En ajoutant des champs dans *manager* ou dans le champs *internal* de *manager* ? [bj] Je suis en faveur de cette solution : les paramètres des heuristiques dépendent fortement du domaine abstrait

41. **Ensemble des options.**

⟨*⟩+≡

```
  /* Options (in) (opaque type) */
  typedef struct ap_option_t {
    ap_funopt_t funopt[AP_FUNID_SIZE];
    bool abort_if_exception[AP_EXC_SIZE];
    ap_scalar_discr_t scalar_discr; /* Preferred type for scalars */
  } ap_option_t;
```

42. **Question.** [bj] Dans les deux types précédents, on a choisi une organisation mémoire à la C, dans laquelle une structure hiérarchique est mise à plat. Ceci est est plus pratique en C, car ne nécessite ni constructeur ni manipulation de pointeurs, mais pour l'interfaçage avec OCaml ou Java, c'est nettement moins pratique.

5.5 Contexte d'appel

Le type `manager_t` est structuré comme suit :

⟨*⟩+≡

```
/* ========================================================================== */
/* I.3 Manager */
/* ========================================================================== */

/* Manager (opaque type) */
typedef struct ap_manager_t {
  char* library;                 /* name of the effective library */
  char* version;                 /* version of the effective library */
  void* internal;                /* library dependent,
                                    should be different for each thread
                                    (working space) */
  void* funptr[AP_FUNID_SIZE];   /* Array of function pointers,
                                    initialized by the effective library */
  struct ap_option_t option;     /* Options (in) */
  struct ap_result_t result;     /* Exceptions and other indications (out) */
  void (*internal_free)(void*); /* deallocation function for internal */
  size_t count;                 /* reference counter */
} ap_manager_t;
```

On utilise du comptage de référence, car le gestionnaire sera référencée depuis les valeurs abstraites de niveau 0 (et plus), et il faut éviter de le désallouer si une de ces valeurs pointe encore sur lui.

43. **Tableau pour les appels virtuels.** Le champ `funptr` du gestionnaire est un tableau de (pointeurs de) fonctions. Ceci permet d'implémenter des fonctions abstraites du type :

⟨*bidon*⟩≡

```
static inline abstract0_t* abstract0_meet(manager_t* man, const abstract0_t* a1, const abstract0_t* a
  abstract0_t* (*ptr)(manager_t*,...) = man->funptr[fun_meet];
  return ptr(man,a1,a2);
}
```

Ceci permet au gestionnaire de jouer le rôle de sélecteur de librairie. Seules les opérations relatives au gestionnaire dépendent de la librairies effectivement utilisées, et toutes les opérations du module `abstract0` sont génériques.

44. **Gestion du gestionnaire (allocation,déallocation).**

le chapitre 11 donne plus de détail là-dessus. Sommairement, chaque librairie interfacée doit fournir une fonction d'alllocation et d'initialisation du gestionnaire. Cette fonction initialise en particulier la table virtuelle et le champ `internal` si la librairie en a l'usage.

5.6 Fonctions d'accès

⟨*⟩+≡

```
/* ********************************************************************** */
/* II. User Functions */
/* ********************************************************************** */

void ap_manager_clear_exclog(ap_manager_t* man);
  /* erase the current log of exception */
void ap_manager_free(ap_manager_t* man);
  /* dereference the counter,
     and possibly free internal field if it is not yet put to NULL */

/* Reading fields */
const char* ap_manager_get_library(ap_manager_t* man);
const char* ap_manager_get_version(ap_manager_t* man);

ap_funopt_t ap_manager_get_funopt(ap_manager_t* man, ap_funid_t funid);
bool ap_manager_get_abort_if_exception(ap_manager_t* man, ap_exc_t exn);

ap_exc_t ap_manager_get_exception(ap_manager_t* man);
  /* Get the last exception raised */
ap_exclog_t* ap_manager_get_exclog(ap_manager_t* man);
  /* Get the full log of exception */
tbool_t ap_manager_get_flag_exact(ap_manager_t* man);
tbool_t ap_manager_get_flag_best(ap_manager_t* man);

/* Settings fields */
void ap_funopt_init(ap_funopt_t* fopt);
void ap_manager_set_funopt(ap_manager_t* man, ap_funid_t funid, ap_funopt_t* funopt);
void ap_manager_set_abort_if_exception(ap_manager_t* man, ap_exc_t exn, bool flag);
```

⟨*⟩+≡

```
/* ********************************************************************** */
/* III. Implementor Functions */
/* ********************************************************************** */

ap_manager_t* ap_manager_alloc(char* library, char* version,
                               void* internal,
                               void (*internal_free)(void*));
static inline
ap_manager_t* ap_manager_copy(ap_manager_t* man);
  /* Increment the reference counter and return its argument */
void ap_manager_raise_exception(ap_manager_t* man,
                                ap_exc_t exn, ap_funid_t funid, const char* msg);
  /* raise an exception and put fiels
     man->result.flag_exact et man->result.flag_best to
     tbool_false
  */
ap_exclog_t* ap_exc_cons(ap_exc_t exn,
                         ap_funid_t funid, const char* msg,
                         ap_exclog_t* tail);
void ap_exclog_free(ap_exclog_t* head);
```

5.7 Définition des fonctions en-ligne

⟨*⟩+≡

```
/* ********************************************************************** */
/* IV. Definition of previously declared inline functions */
/* ********************************************************************** */

static inline tbool_t tbool_of_int(int n)
{ return ((n)? tbool_true : tbool_false); }
static inline tbool_t tbool_of_bool(bool a)
  { return ((a)? tbool_true : tbool_false); }
static inline tbool_t tbool_or(tbool_t a, tbool_t b)
{
  return
    ( (a==tbool_true || b==tbool_true)? tbool_true :
      ( (a==tbool_top || b==tbool_top)? tbool_top :
        tbool_false ) );
}
static inline tbool_t tbool_and(tbool_t a, tbool_t b)
{
  return
    ( (a==tbool_false || b==tbool_false)? tbool_false :
      ( (a==tbool_top || b==tbool_top)? tbool_top :
        tbool_true ) );
}
static inline
ap_manager_t* ap_manager_copy(ap_manager_t* man)
{ man->count++; return man; }
```

⟨*⟩+≡
```
#ifdef __cplusplus
}
#endif

#endif
```

Chapitre 6

Interface du domaine abstrait : fichier ap_abstract0.h

⟨*⟩≡

```
/* ********************************************************************** */
/* ap_abstract0.h : generic operations on numerical abstract values */
/* ********************************************************************** */

/* This file is part of the APRON Library, released under LGPL license.  Please
   read the COPYING file packaged in the distribution */

/* GENERATED FROM ap_abstract0.nw : DO NOT MODIFY! */


#ifndef _AP_ABSTRACT0_H_
#define _AP_ABSTRACT0_H_

#include "ap_manager.h"
#include "ap_expr0.h"

#ifdef __cplusplus
extern "C" {
#endif

/* Generic abstract value at level 0 */
typedef struct ap_abstract0_t {
  void* value;    /* Abstract value of the underlying library */
  ap_manager_t* man; /* Used to identify the effective type of value */
} ap_abstract0_t;
```

6.1 Gestion mémoire, Représentations, Entrées/Sorties

⟨*⟩+≡

```
/* ********************************************************************** */
/* I. General management */
/* ********************************************************************** */
```

45. **Initialisation.** Elle se fait en allouant un manager.

46. **Gestion mémoire**

⟨*⟩+≡

```
/* ============================================================ */
/* I.1 Memory */
/* ============================================================ */

ap_abstract0_t* ap_abstract0_copy(ap_manager_t* man, const ap_abstract0_t* a);
  /* Return a copy of an abstract value, on
     which destructive update does not affect the initial value. */

void ap_abstract0_free(ap_manager_t* man, ap_abstract0_t* a);
  /* Free all the memory used by the abstract value */

size_t ap_abstract0_size(ap_manager_t* man, const ap_abstract0_t* a);
  /* Return the abstract size of an abstract value (see ap_manager_t) */
```

47. **Control of internal representation**

⟨*⟩+≡

```
/* ============================================================ */
/* I.2 Control of internal representation */
/* ============================================================ */


void ap_abstract0_minimize(ap_manager_t* man, const ap_abstract0_t* a);
  /* Minimize the size of the representation of a.
     This may result in a later recomputation of internal information.
  */


void ap_abstract0_canonicalize(ap_manager_t* man, const ap_abstract0_t* a);
  /* Put the abstract value in canonical form. (not yet clear definition) */


void ap_abstract0_approximate(ap_manager_t* man, ap_abstract0_t* a, int algorithm);
  /* Perform some transformation on the abstract value, guided by the
     field algorithm.

     The transformation may lose information.  The argument "algorithm"
     overrides the field algorithm of the structure of type ap_funopt_t
     associated to ap_abstract0_approximate (commodity feature). */


tbool_t ap_abstract0_is_minimal(ap_manager_t* man, const ap_abstract0_t* a);

tbool_t ap_abstract0_is_canonical(ap_manager_t* man, const ap_abstract0_t* a);
```

Les propriétés de la forme canonique ne sont pas encore bien claires (*c.f.* §1.4). On aimerait que par exemple la sérialisation en binaire d'un objet canonique soit aussi canonique.

48. **Impression.**

⟨*⟩+≡

```
/* ============================================================ */
/* I.3 Printing */
/* ============================================================ */


void ap_abstract0_fprint(FILE* stream,
                         ap_manager_t* man,
                         const ap_abstract0_t* a,
                         char** name_of_dim);
  /* Print the abstract value in a pretty way, using function
     name_of_dim to name dimensions */


void ap_abstract0_fprintdiff(FILE* stream,
                             ap_manager_t* man,
                             const ap_abstract0_t* a1, const ap_abstract0_t* a2,
                             char** name_of_dim);
  /* Print the difference between a1 (old value) and a2 (new value),
     using function name_of_dim to name dimensions.
     The meaning of difference is library dependent. */


void ap_abstract0_fdump(FILE* stream, ap_manager_t* man, const ap_abstract0_t* a);
  /* Dump the internal representation of an abstract value,
     for debugging purposes */
```

49. **Précisions sur les fonctions d'impressions.** Le format des fonctions d'impression est propre à chaque librairie : aucune syntaxe n'est imposée, ne serait-ce que parce que la représentation interne des nombres n'est pas unifiée.

Si l'on désire une syntaxe uniforme, il faut d'abord convertir en contraintes/générateurs utilisateur.

??

50. **Sérialisation**

La sérialisation/désérialisation en binaire, comme déjà mentionné, ne fonctionne que pour la même librairie, compilée avec les mêmes options de représentation interne des nombres.

Antoine Miné, dans ses octogones, a des fonctions de sérialisation vers des zones mémoires (au lieu d'un fichier). C'est plus général, et nécessaire, par exemple pour interfacer avec OCaml. On ne propose donc que la sérialisation vers des zones mémoires, l'utilisateur C pourra ensuite utiliser les fonctions `fwrite` et `fread` pour transférer des zones mémoire de et vers un flux entrée/sortie..

⟨*⟩+≡

```
/* ============================================================ */
/* I.4 Serialization */
/* ============================================================ */


ap_membuf_t ap_abstract0_serialize_raw(ap_manager_t* man, const ap_abstract0_t* a);
/* Allocate a memory buffer (with malloc), output the abstract value in raw
   binary format to it and return a pointer on the memory buffer and the size
   of bytes written.  It is the user responsability to free the memory
   afterwards (with free). */


ap_abstract0_t* ap_abstract0_deserialize_raw(ap_manager_t* man, void* ptr, size_t* size);
/* Return the abstract value read in raw binary format from the input stream
   and store in size the number of bytes read */
```

6.2 Constructeurs, accesseurs, tests et extraction de propriétés

51. **Basic constructors**

⟨*⟩+≡

```
/* ********************************************************************** */
/* II. Constructor, accessors, tests and property extraction */
/* ********************************************************************** */

/* ============================================================ */
/* II.1 Basic constructors */
/* ============================================================ */

/* We assume that dimensions [0..intdim-1] correspond to integer variables, and
   dimensions [intdim..intdim+realdim-1] to real variables */

ap_abstract0_t* ap_abstract0_bottom(ap_manager_t* man, size_t intdim, size_t realdim);
  /* Create a bottom (empty) value */


ap_abstract0_t* ap_abstract0_top(ap_manager_t* man, size_t intdim, size_t realdim);
  /* Create a top (universe) value */
```

On pourrait aussi rajouter les deux fonctions suivantes (qui peuvent être vues comme des fonctions de conversion).

⟨*⟩+≡

```
ap_abstract0_t* ap_abstract0_of_box(ap_manager_t* man,
                                    size_t intdim, size_t realdim,
                                    const ap_interval_t*const* tinterval);
  /* Abstract an hypercube defined by the array of intervals
     of size intdim+realdim */


ap_abstract0_t* ap_abstract0_of_lincons_array(ap_manager_t* man,
                                              size_t intdim, size_t realdim,
                                              const ap_lincons0_array_t* array);
  /* Abstract a convex polyhedra defined by the array of (interval) linear
     constraints of size size */
```

`ap_abstract0_of_box` permet d'abstraire assez aisément un point de l'espace.

`ap_abstract0_of_lincons_array` est redondant si l'on dispose de `ap_abstract0_meet_lincons_array` (ou autre nom, `ap_abstract0_meet_polyhedra`), puisqu'il suffit alors de créer la valeur top puis d'appeler `ap_abstract0_meet_lincons_array` avec l'ensemble des contraintes.

52. Accessors

⟨*⟩+≡

```
/* ============================================================ */
/* II.2 Accessors */
/* ============================================================ */

ap_dimension_t ap_abstract0_dimension(ap_manager_t* man, const ap_abstract0_t* a);
  /* Return the dimensionality of the abstract values */
```

53. **Tests**

⟨*⟩+≡

```
/* ============================================================ */
/* II.3 Tests */
/* ============================================================ */

/* If any of the following functions returns tbool_top, this means that
   an exception has occured, or that the exact computation was
   considered too expensive to be performed (according to the options).
   The flag exact and best should be cleared in such a case. */


tbool_t ap_abstract0_is_bottom(ap_manager_t* man, const ap_abstract0_t* a);

tbool_t ap_abstract0_is_top(ap_manager_t* man, const ap_abstract0_t* a);


tbool_t ap_abstract0_is_leq(ap_manager_t* man, const ap_abstract0_t* a1, const ap_abstract0_t* a2);
  /* inclusion check */


tbool_t ap_abstract0_is_eq(ap_manager_t* man, const ap_abstract0_t* a1, const ap_abstract0_t* a2);
  /* equality check */


tbool_t ap_abstract0_sat_lincons(ap_manager_t* man, const ap_abstract0_t* a, const ap_lincons0_t* lin
  /* does the abstract value satisfy the (interval) linear constraint? */


tbool_t ap_abstract0_sat_interval(ap_manager_t* man, const ap_abstract0_t* a,
                                  ap_dim_t dim, const ap_interval_t* interval);
  /* is the dimension included in the interval in the abstract value? */


tbool_t ap_abstract0_is_dimension_unconstrained(ap_manager_t* man, const ap_abstract0_t*
                                                a, ap_dim_t dim); /* is the dimension
                                                unconstrained in the abstract
                                                value?  If it is the case, we
                                                have forget(man,a,dim) == a */
```

54. **Le problème du test du vide avec les entiers.** Que doit retourner `ap_abstract0_is_bottom` si les entiers ne sont que partiellement pris en compte ? Si un polyèdre n'est pas vide en rationnel, on peut retourner :

- `tbool_false` avec le flag `flag_exact` à `tbool_false` ou `tbool_top` ;
- `tbool_true` avec le flag `flag_exact` à `tbool_false` ou `tbool_top` ;
- `tbool_top` avec le flag `flag_exact` à n'importe quelle valeur
- ...

En fait, la redondance illustrée ci-dessus entre la valeur retournée et le flag `flag_exact` est générale à tous les tests.

Comment faire si on veut juste savoir si un polyèdre sur des dimensions déclarées entières est vide ou non au sens des rationnels ? Dans NewPolka, on utilise quand-même le typage des dimensions pour nier les contraintes, mais pour le reste on ne considère que des rationnels. Bref, sur les expressions, on type en entier, mais sur les polyèdres, on type en réel.

55. **Extraction de propriétés**

⟨*⟩+≡

```
/* ========================================================== */
/* II.4 Extraction of properties */
/* ========================================================== */


ap_interval_t* ap_abstract0_bound_linexpr(ap_manager_t* man,
                                    const ap_abstract0_t* a, const ap_linexpr0_t* expr);
  /* Returns the interval taken by a linear expression
     over the abstract value */


ap_interval_t* ap_abstract0_bound_dimension(ap_manager_t* man,
                                      const ap_abstract0_t* a, ap_dim_t dim);
  /* Returns the interval taken by the dimension
     over the abstract value */


ap_lincons0_array_t ap_abstract0_to_lincons_array(ap_manager_t* man, const ap_abstract0_t* a);
  /* Converts an abstract value to a polyhedra
     (conjunction of linear constraints).

     The constraints are normally guaranteed to be really linear (without intervals) */


ap_interval_t** ap_abstract0_to_box(ap_manager_t* man, const ap_abstract0_t* a);
  /* Converts an abstract value to an interval/hypercube.
     The size of the resulting array is ap_abstract0_dimension(man,a).  This
     function can be reimplemented by using ap_abstract0_bound_linexpr */


ap_generator0_array_t ap_abstract0_to_generator_array(ap_manager_t* man, const ap_abstract0_t* a);
  /* Converts an abstract value to a system of generators. */
```

6.3 Opérations

La plupart des opérations sont offertes en 2 versions : fonctionnelle ou destructrice. On utilise un booléen por distinguer les 2 sémantiques. La sémantique destructrice est la suivante : après l'appel, la première valeur abstraite de la fonction est détruite, et ne doit plus être manipulée ni référencée. Il est possible que la valeur retournée par la fonction soit la même que l'argument, mais il faut la manipuler via la valeur retournée.

56. Bornes supérieures et inférieures

⟨*⟩+≡

```
/* ********************************************************************** */
/* III. Operations : functional version */
/* ********************************************************************** */

/* ============================================================ */
/* III.1 Meet and Join */
/* ============================================================ */


ap_abstract0_t* ap_abstract0_meet(ap_manager_t* man, bool destructive, ap_abstract0_t* a1, const ap_a

ap_abstract0_t* ap_abstract0_join(ap_manager_t* man, bool destructive, ap_abstract0_t* a1, const ap_a
  /* Meet and Join of 2 abstract values */


ap_abstract0_t* ap_abstract0_meet_array(ap_manager_t* man, const ap_abstract0_t*const* tab, size_t si

ap_abstract0_t* ap_abstract0_join_array(ap_manager_t* man, const ap_abstract0_t*const* tab, size_t si
  /* Meet and Join of an array of abstract values.
     Raises an [[exc_invalid_argument]] exception if [[size==0]]
     (no way to define the dimensionality of the result in such a case */


ap_abstract0_t* ap_abstract0_meet_lincons_array(ap_manager_t* man,
                                               bool destructive,
                                               ap_abstract0_t* a,
                                               const ap_lincons0_array_t* array);
  /* Meet of an abstract value with a set of constraints */


ap_abstract0_t* ap_abstract0_add_ray_array(ap_manager_t* man,
                                          bool destructive,
                                          ap_abstract0_t* a,
                                          const ap_generator0_array_t* array);
  /* Generalized time elapse operator */
```

Discussion sur `ap_abstract0_meet_lincons_array` :
- Plus général que `ap_abstract0_of_lincons_array`, *c.f.* §10.3.
- Même si implanté par appel successifs à `ap_abstract0_meet_lincons`, les appels intermédiaires peuvent utiliser des effets de bord.
- Dans le cas des polyèdres, plus efficace que des appels successifs, et plus efficace qu'appeler `ap_abstract0_of_lincons_array` et faire l'intersection.

57. **Affectations et Substitutions.**

Il a été décidé de ne pas implanter les *weak update*. D'après Antoine Miné, cela ne va pas assez loin. D'apères Bertrand Jeannet,, Reps, Sagiv & al ne l'utilise pas. Les *weak update* peuvent par ailleurs être implantés à partir des opérations ci-dessus. En revanche, l'expansion et le pliage de dimension est fourni pour autoriser la notion de *weak update* à un niveau supérieur, voir la suite.

⟨*⟩+≡

```
/* ========================================================== */
/* III.2 Assignement and Substitutions */
/* ========================================================== */


ap_abstract0_t* ap_abstract0_assign_linexpr(ap_manager_t* man,
                                    bool destructive,
                                    ap_abstract0_t* org,
                                    ap_dim_t dim, const ap_linexpr0_t* expr,
                                    const ap_abstract0_t* dest);

ap_abstract0_t* ap_abstract0_substitute_linexpr(ap_manager_t* man,
                                        bool destructive,
                                        ap_abstract0_t* org,
                                        ap_dim_t dim, const ap_linexpr0_t* expr,
                                        const ap_abstract0_t* dest);
  /* Assignement and Substitution of a single dimension by a (interval)
     linear expression in abstract value org.

     dest is an optional argument. If not NULL, semantically speaking,
     the result of the transformation is intersected with dest. This is
     useful for precise backward transformations in lattices like intervals or
     octagons.
 */


ap_abstract0_t* ap_abstract0_assign_linexpr_array(ap_manager_t* man,
                                          bool destructive,
                                          ap_abstract0_t* org,
                                          const ap_dim_t* tdim,
                                          const ap_linexpr0_t*const* texpr,
                                          size_t size,
                                          const ap_abstract0_t* dest);

ap_abstract0_t* ap_abstract0_substitute_linexpr_array(ap_manager_t* man,
                                              bool destructive,
                                              ap_abstract0_t* org,
                                              const ap_dim_t* tdim,
                                              const ap_linexpr0_t*const* texpr,
```

```
                                                size_t size,
                                                const ap_abstract0_t* dest);
  /* Parallel Assignement and Substitution of several dimensions by
     linear expressions in abstract value org.

     dest is an optional argument. If not NULL, semantically speaking,
     the result of the transformation is intersected with dest. This is
     useful for precise backward transformations in lattices like intervals or
     octagons. */
```

58. **Projections et Quantifications existentielles.**

⟨*⟩+≡

```
  /* ============================================================ */
  /* III.3 Projections */
  /* ============================================================ */

  ap_abstract0_t* ap_abstract0_forget_array(ap_manager_t* man,
                                    bool destructive,
                                    ap_abstract0_t* a, const ap_dim_t* tdim, size_t size,
                                    bool project);
```

Questions :

- Peut-on distinguer `projet` de `forget` juste avec un flag histoire de diminuer le nombre de fonctions ? + voir remarque suivante.
- Dans tous les treillis envisagés, `project` peut s'implanter à partir de `forget` suivi d'une affectation à zéro. La réciproque est vraie, en utilisant `add_ray_array`, mais c'est sans doute plus compliqué pour pas mal de treillis.
- Nommage de `forget` : on peut aussi l'appeler `exist`. De manière générale, doit-on utiliser un vocabulaire mathématique (`exist`, `least upper bound`, ...) ou le vocabulaire d'usage, notamment outre-atlantique (`forget`, `join`, ...) ?

59. **Changement et Permutations de dimensions.**

On fournit seulement des fonctions générales, permettant d'effectuer des permutations et nécessaires pour le niveau 1 de l'interface.

On pourra éventuellement ajouter des fonctions plus simples, mais peut-on espérer un gain en efficacité réel ? Sinon, comme on ne se préoccupe pas du confort à ce niveau, autant en rester là.

Question identique à celle du paragraphe précédent : utilise-t-on un flag pour distinguer `project` de `forget` ou `embed` ?

⟨*⟩+≡

```
/* ============================================================ */
/* III.4 Change and permutation of dimensions */
/* ============================================================ */

ap_abstract0_t* ap_abstract0_add_dimensions(ap_manager_t* man,
                                        bool destructive,
                                        ap_abstract0_t* a,
                                        ap_dimchange_t* dimchange,
                                        bool project);
ap_abstract0_t* ap_abstract0_remove_dimensions(ap_manager_t* man,
                                           bool destructive,
                                           ap_abstract0_t* a,
                                           ap_dimchange_t* dimchange);
/* Size of the permutation is supposed to be equal to
   the dimension of the abstract value */
ap_abstract0_t* ap_abstract0_permute_dimensions(ap_manager_t* man,
                                            bool destructive,
                                            ap_abstract0_t* a,
                                            const ap_dimperm_t* perm);
```

60. **Expansion et pliage**

Il a été décidé de fournir au niveau 0 un support pour l'expansion et le pliage de dimensions.

Formellement, la sémantique concrète est la suivante (sachant que ces opérations sont associatives, et que z est expansé en z et w, puis z et w plié sur z) :

$$\mathsf{expand}(P(x,y,z),z,w) = P(x,y,z) \wedge P(x,y,w) \tag{6.1}$$

$$\mathsf{fold}((Q(x,y,z,w),z,w) = (\exists w : Q(x,y,z,w)) \vee (\exists z : Q(x,y,z,w))[z \leftarrow w] \tag{6.2}$$

Là encore, ces opérations ne sont pas minimales, mais pour des raisons de performances il semble que ça vaille le coup de fournir des opérations spécifiques.

⟨*⟩+≡

```
/* ========================================================== */
/* III.5 Expansion and folding of dimensions */
/* ========================================================== */


ap_abstract0_t* ap_abstract0_expand(ap_manager_t* man,
                                    bool destructive,
                                    ap_abstract0_t* a,
                                    ap_dim_t dim,
                                    size_t n);
  /* Expand the dimension dim into itself + n additional dimensions.
     It results in (n+1) unrelated dimensions having same
     relations with other dimensions. The (n+1) dimensions are put as follows :

     - original dimension dim

     - if the dimension is integer, the n additional dimensions are put at the
       end of integer dimensions; if it is real, at the end of the real
       dimensions.
  */


ap_abstract0_t* ap_abstract0_fold(ap_manager_t* man,
                                  bool destructive,
                                  ap_abstract0_t* a,
                                  const ap_dim_t* tdim,
                                  size_t size);
  /* Fold the dimensions in the array tdim of size n>=1 and put the result
     in the first dimension in the array. The other dimensions of the array
     are then removed (using ap_abstract0_permute_remove_dimensions). */
```

61. **Élargissement.**

⟨*⟩+≡

```
/* ========================================================== */
/* III.6 Widening */
/* ========================================================== */

ap_abstract0_t* ap_abstract0_widening(ap_manager_t* man,
                                      const ap_abstract0_t* a1, const ap_abstract0_t* a2);
```

62. **Clôture topologique.**

Il faut décider si l'on traite les contraintes strictes. [bj] Ça ne mange pas de pain, il reste toujours la possibilité d'ignorer le caractère stricte d'une contrainte pour une implantation particulière.

⟨*⟩+≡

```
/* ========================================================== */
/* III.7 Closure operation */
/* ========================================================== */

/* Returns the topological closure of a possibly opened abstract value */


ap_abstract0_t* ap_abstract0_closure(ap_manager_t* man, bool destructive, ap_abstract0_t* a);
```

6.4 Fonctions communes aux niveau 0

On définit ici les fonctions qui n'ont pas d'équivalent dans les librairies sous-jacentes, et qui sont factorisées entre ces librairies.

⟨*⟩+≡

```
/* ********************************************************************** */
/* ********************************************************************** */
/* Additional functions */
/* ********************************************************************** */
/* ********************************************************************** */
static inline
ap_manager_t* ap_abstract0_manager(const ap_abstract0_t* a)
  { return a->man; }
  /* Return a reference to the manager contained in the abstract value.
     The reference should not be freed */

/* Widening with threshold */

ap_abstract0_t* ap_abstract0_widening_threshold(ap_manager_t* man,
                                const ap_abstract0_t* a1, const ap_abstract0_t* a2,
                                ap_lincons0_array_t* array);

ap_linexpr0_t* ap_abstract0_quasilinear_of_intervallinear(ap_manager_t* man,
                                                          const ap_abstract0_t* a,
                                                          ap_linexpr0_t* expr);
  /* Evaluate a interval linear expression on the abstract
     value such as to transform it into a quasilinear expression.

     This implies calls to ap_abstract0_bound_dimension.

     NOT YET IMPLEMENTED
  */
```

6.5 Fonctions internes pour le niveau 1

On définit ici des fonctions utiles pour le niveau 1.

⟨*⟩+≡
```
/* ********************************************************************** */
/* ********************************************************************** */
/* Internal functions */
/* ********************************************************************** */
/* ********************************************************************** */

ap_abstract0_t* ap_abstract0_meetjoin(ap_funid_t funid,
                                      ap_manager_t* man, bool destructive,
                                      ap_abstract0_t* a1, const ap_abstract0_t* a2);
ap_abstract0_t* ap_abstract0_asssub_linexpr(ap_funid_t funid,
                                            /* either assign or substitute */
                                            ap_manager_t* man,
                                            bool destructive,
                                            ap_abstract0_t* a,
                                            ap_dim_t dim, const ap_linexpr0_t* expr,
                                            const ap_abstract0_t* dest);
ap_abstract0_t* ap_abstract0_asssub_linexpr_array(ap_funid_t funid,
                                                  ap_manager_t* man,
                                                  bool destructive,
                                                  ap_abstract0_t* a,
                                                  const ap_dim_t* tdim,
                                                  const ap_linexpr0_t*const* texpr,
                                                  size_t size,
                                                  const ap_abstract0_t* dest);
```

⟨*⟩+≡
```
#ifdef __cplusplus
}
#endif

#endif
```

Chapitre 7

Var : définition et manipulation de variables génériques, fichier ap_var.h

⟨*⟩≡
```
/* ********************************************************************** */
/* ap_var.h : variables (strings or user-defined) */
/* ********************************************************************** */

/* This file is part of the APRON Library, released under LGPL license.  Please
   read the COPYING file packaged in the distribution */

/* GENERATED FROM ap_var.nw : DO NOT MODIFY! */

#ifndef _AP_VAR_H_
#define _AP_VAR_H_

#ifdef __cplusplus
extern "C" {
#endif
```

7.1 Datatypes

⟨*⟩+≡

```
/* ========================================================================== */
/* Datatype */
/* ========================================================================== */

/* The abstract type ap_var_t is
   equipped with a total ordering function, a copy function,
   and a free function.

   The parametrization is done via a global variable pointing to an
   ap_var_operations_t structure.
 */

typedef void* ap_var_t;

typedef struct ap_var_operations_t {
  int (*compare)(ap_var_t v1, ap_var_t v2); /* Total ordering function */
  ap_var_t (*copy)(ap_var_t var);           /* Duplication function */
  void (*free)(ap_var_t var);               /* Deallocation function */
  char* (*to_string)(ap_var_t var);         /* Conversion to a dynamically allocated string */
} ap_var_operations_t;

extern ap_var_operations_t ap_var_operations_default;
  /* default manager : ap_var_t are char* */

extern ap_var_operations_t* ap_var_operations;
/* points to manager in use, by default ap_var_operations_default */

#ifdef __cplusplus
}
#endif

#endif
```

Chapitre 8

Environment : liaison des variables aux dimensions (entiers), fichier ap_environlent.h

Ce fichier définit les environnements qui font le lien avec d'une part des *variables* et d'autre part leur type et la dimension qui leur est associée au niveau 0.

On distingue deux catégories de fonctions :

- Celles utiles à un utilisateur du niveau 1 de l'interface, pour manipuler des environnements. Il s'agit des fonctions `ap_environment_free, ap_environment_alloc, ap_environment_add, ap_environment_remove, ap_environment_is_eq, environement_is_leq, ap_environment_compare`.
- Celles utilisées essentiellement par l'interface commune (module abstract1) pour passer automatiquement du niveau 1 au niveau 0, qui sont d'un usage interne

⟨*⟩≡

```
/* ********************************************************************** */
/* ap_environment.h : binding of addresses (strings) to dimensions */
/* ********************************************************************** */

/* This file is part of the APRON Library, released under LGPL license.  Please
   read the COPYING file packaged in the distribution */

/* GENERATED FROM ap_environment.nw : DO NOT MODIFY! */

#ifndef _AP_ENVIRONMENT_H_
#define _AP_ENVIRONMENT_H_

#include "ap_config.h"
#include "ap_dimension.h"
#include "ap_var.h"

#ifdef __cplusplus
extern "C" {
#endif
```

8.1 Datatypes

⟨*⟩+≡

```
/* ========================================================================== */
/* Datatype */
/* ========================================================================== */
```

⟨*⟩+≡

```
/* The ap_environment_t type is the type used by the level 1 of the interface.
   It should be considered as an abstract type.

   We use reference counting to manage memory. Conventions are :

   - a newly allocated environement has a reference count of one ;
   - environement_copy increments the counter and return its argument
   - environement_free decrements it and free the environement
     in case of zero or negative number.
*/

typedef struct ap_environment_t {
  ap_var_t* var_of_dim;
  /*
    Array of size intdim+realdim, indexed by dimensions.
    - It should not contain identical strings..
    - Slice [0..intdim-1] is lexicographically sorted,
      and denotes integer variables.
    - Slice [intdim..intdim+realdim-1] is lexicographically sorted,
      and denotes real variables.
    - The memory allocated for the variables are attached to the structure
      (they are freed when the structure is no longer in use)
  */
  size_t intdim; /* Number of integer variables */
  size_t realdim;/* Number of real variables */
  size_t count; /* For reference counting */
} ap_environment_t;

typedef struct ap_environment_name_of_dim_t {
  size_t size;
  char* p[] ; /* Assumed to be of size size */
} ap_environment_name_of_dim_t;
```

8.2 Memory management, Constructor, Destructors

⟨*⟩+≡

```
  /* ========================================================================== */
  /* Memory management, Constructor, Destructors */
  /* ========================================================================== */

  void ap_environment_free2(ap_environment_t* e) ;
    /* Free the environement
       (the structure itself and the memory pointed to by fields) */

  static inline
  void ap_environment_free(ap_environment_t* e) ;
    /* Decrement the reference counter and possibly deallocate */

  static inline
  ap_environment_t* ap_environment_copy(ap_environment_t* e) ;
    /* Copy */

  void ap_environment_fdump(FILE* stream, ap_environment_t* env) ;
    /* Print an environement under the form :
        environment : dim = (..,..), count = ..
        0 : name0
        1 : name1
        ...
    */

  ap_environment_name_of_dim_t* ap_environment_name_of_dim_alloc(ap_environment_t* e) ;
  void ap_environment_name_of_dim_free(ap_environment_name_of_dim_t*) ;

  ap_environment_t* ap_environment_alloc_empty(void) ;
    /* Build an empty environement */

  ap_environment_t* ap_environment_alloc(ap_var_t* name_of_intdim, size_t intdim,
                                         ap_var_t* name_of_realdim, size_t realdim) ;
    /* Build an environement from an array of integer and
       an array of real variables.
       - Variables are duplicated in the result, so it is the responsability
         of the user to free the variables he provides.
       - If the result does not satisfy the invariant, return NULL.
    */

  ap_environment_t* ap_environment_add(const ap_environment_t* env,
                                       ap_var_t* name_of_intdim, size_t intdim,
                                       ap_var_t* name_of_realdim, size_t realdim) ;
    /* Add variables to an environement.
       Same comments as for ap_environment_alloc */
  ap_environment_t* ap_environment_add_perm(const ap_environment_t* env,
                                            ap_var_t* name_of_intdim, size_t intdim,
                                            ap_var_t* name_of_realdim, size_t realdim,
                                            ap_dimperm_t* dimpermu) ;
    /* Same as previous functions, but in addition return in *dimperm
       the permutation to apply after having added new variables at the end of their
       respective slice, in order to get them sorted.
       If the result is NULL, so is dimperm->dim */
  ap_environment_t* ap_environment_remove(const ap_environment_t* env,
```

```
                                    ap_var_t* name_of_intdim, size_t intdim,
                                    ap_var_t* name_of_realdim, size_t realdim);
  /* Remove variables from an environement.
     Same comments as for environement_alloc */

ap_dim_t ap_environment_dim_of_var(const ap_environment_t* env, ap_var_t name);
  /* - If the variable is present in the environemnt,
       return its associated dimension.
     - Otherwise, return AP_DIM_MAX
  */

static inline
ap_var_t ap_environment_var_of_dim(const ap_environment_t* env, ap_dim_t dim);
  /* - Return the variable associated to the dimension.
     - There is no bound check here */
```

8.3 Tests

⟨*⟩+≡

```
/* ============================================================================ */
/* Tests */
/* ============================================================================ */

bool ap_environment_is_eq(const ap_environment_t* env1,
                          const ap_environment_t* env2);
  /* Equality test */
bool ap_environment_is_leq(const ap_environment_t* env1,
                          const ap_environment_t* env2);
  /* Inclusion test */
int ap_environment_compare(const ap_environment_t* env1,
                           const ap_environment_t* env2);
  /* Return
    - -2 if the environements are not compatible
      (a variable has a different type in the 2 environements)
    - -1 if env1 is a subset of env2
    - 0 if equality
    - +1 if env1 is a superset of env2
    - +2 otherwise (the lce exists and is a strict superset of both)
  */
```

8.4 Least common environments and conversion permutations

⟨*⟩+≡

```
/* ========================================================================== */
/* Least common environments and conversion permutations */
/* ========================================================================== */

ap_dimchange_t* ap_environment_dimchange(const ap_environment_t* env1,
                                         const ap_environment_t* env);
  /* Compute the transformation for converting from an environment to a
     superenvironment.
     Return NULL if env is not a superenvironment.
  */

ap_environment_t* ap_environment_lce(const ap_environment_t* env1,
                                     const ap_environment_t* env2,
                                     ap_dimchange_t** dimchange1,
                                     ap_dimchange_t** dimchange2);
  /*
  Least common environment to two environements.

   - If environments are not compatible
     (a variable has different types in the 2 environements),
     return NULL

   - Compute also in dimchange1 and dimchange2 the conversion transofrmations.

   - If no dimensions to add to env1, this implies that env is
     actually env1. In this case, *dimchange1==NULL. Otherwise, the
     function allocates *dimchange1 with ap_dimchange_alloc.
 */
ap_environment_t* ap_environment_lce_array(const ap_environment_t** tenv,
                                           size_t size,
                                           ap_dimchange_t*** ptdimchange);
/*
  Least common environement to an array environements.

  - Assume the size of the array is at least one.

  - If all input environements are the same, *ptdimchange==NULL.
    Otherwise, compute in *ptdimchange the conversion permutations.

  - If no dimensions to add to tenv[i], this implies that env is actually
    tenv[i]. In this case, (*ptdimchange)[i]==NULL.
    Otherwise, the function allocates
    (*ptdimchange)[0] with ap_dimchange_alloc.

 */
```

8.5 Renommage de variables

⟨*⟩+≡

```
  /* ============================================================================ */
  /* Variable renaming */
  /* ============================================================================ */

  ap_environment_t* ap_environment_rename(ap_environment_t* env,
                                          ap_var_t* tvar1, ap_var_t* tvar2, size_t size,
                                          ap_dimperm_t* perm);
    /* Rename the variables in the environment.
       size is the common size of tvar1 and tvar2,
       and perm is a result-parameter

       The function applies the variable substitution tvar1[i]->tvar2[i] to
       the environment, and returns the resulting environment and
       the allocated transformation permutation in *permutation.

       If the parameters are not valid, returns NULL with perm->dim==NULL
    */
```

8.6 Définitions en-ligne

⟨*⟩+≡

```
  /* ============================================================================ */
  /* Inline definitions */
  /* ============================================================================ */

  static inline
  ap_var_t ap_environment_var_of_dim(const ap_environment_t* env, ap_dim_t dim){
    return dim < env->intdim+env->realdim? env->var_of_dim[dim] : NULL;
  }
  static inline
  void ap_environment_free(ap_environment_t* env){
    if (env->count<=1)
      ap_environment_free2(env);
    else
      env->count-;
  }
  static inline
  ap_environment_t* ap_environment_copy(ap_environment_t* env){
    env->count++;
    return env;
  }
```

⟨*⟩+≡

```
  #ifdef __cplusplus
  }
  #endif

  #endif
```

Chapitre 9

Sémantique concrète niveau 1 et fichier ap_expr1.h

⟨*⟩≡
```
/* ********************************************************************** */
/* ap_expr1.h : datatypes for dimensions, expressions and constraints */
/* ********************************************************************** */

/* This file is part of the APRON Library, released under LGPL license.  Please
   read the COPYING file packaged in the distribution */

/* GENERATED FROM ap_expr1.nw : DO NOT MODIFY! */


#ifndef _AP_EXPR1_H_
#define _AP_EXPR1_H_

#include "ap_linexpr1.h"
#include "ap_lincons1.h"
#include "ap_generator1.h"

#endif
```

9.1 Types de donnée

63. **Principe.** Il est de rajouter un environment au structures de donnée de niveau 0. exemple :
⟨*bidon*⟩≡
```
typedef struct ap_linexpr1_t {
  ap_linexpr0_t* linexpr0;
  environment_t* env;
} ap_linexpr1_t;
typedef struct ap_lincons1_t {
  ap_lincons0_t lincons0;
  environment_t* env;
} ap_lincons1_t;
```

Cela rend très rapide la conversion du niveau 1 au niveau 0 : on sélectionne le champ `linexpr0` ou `lincons0`.

En revanche, si on veut obtenir l'expression de niveau 1 associée à la contrainte, il faut écrire :

⟨*bidon*⟩+≡
```
ap_linexpr1_t ap_linexpr1_of_lincons1(ap_lincons1_t* cons1){
  ap_linexpr1_t expr1;
  expr1.linexpr0 = cons1->lincons0.linexpr0;
  expr1.env = cons1->env;
  return expr1;
}
```

64. **Cas particulier des tableaux.** L'inconvénient de la solution esquissée ci-dessus est accru pour les tableaux :

⟨*bidon*⟩+≡
```
typedef struct ap_ap_lincons1_array_t {
  ap_ap_lincons0_array_t ap_lincons0_array;
  environment_t* env;
} ap_ap_lincons1_array_t;
```

Retourner la contrainte de niveau 1 correspondant à l'index `index` d'un tableau se définit alors par

⟨*bidon*⟩+≡
```
ap_lincons1_t ap_ap_lincons1_array_get(const ap_ap_lincons1_array_t* array,
                                       size_t index){
  ap_lincons1_t cons;
  cons.lincons0 = array->ap_lincons0_array.p[index];
  cons.env = array->env;
  return cons;
}
```

L'affectation de l'index `index` donne :

⟨*bidon*⟩+≡
```
bool ap_ap_lincons1_array_set(ap_ap_lincons1_array_t* array,
                          size_t index, const ap_lincons1_t* cons)
{
  if (index>=array->ap_lincons0_array.size || cons->env != array->env)
    return true;
  array->ap_lincons0_array.p[index] = cons->lincons0;
  environment_deref(array->env);
  return false;
}
```

Il y a cependant un avantage : les structures sont construites une fois, mais utilisées plusieurs fois lors d'une analyse (si on procède de manière classique en construisant à l'avance le système d'équations).

65. **Principe alternatif.** La solution alternative consisterait à écrire

⟨*bidon*⟩+≡
```
typedef struct ap_linexpr1_t {
  ap_linexpr0_t* linexpr0;
  environment_t* env;
} ap_linexpr1_t;
typedef struct ap_lincons1_t {
  ap_linexpr1_t linexpr1;
  ap_constyp_t constyp;
} ap_lincons1_t;
```

Pour convertir du niveau 1 au niveau 0 :

⟨*bidon*⟩+≡
```
ap_lincons0_t ap_lincons0_of_lincons1(ap_lincons1_t* cons1){
  ap_lincons0_t cons0;
  cons0.linexpr0 = cons1->linexpr1.linexpr0;
  cons0.constyp = cons1->constyp;
  return cons0;
}
environment_of_lincons1(ap_lincons1_t* cons1){
  return cons1->linexpr1.env;
}
```

Pour extraire l'expression de niveau 1 contenue dans une contrainte de niveau 1, immédiat. A ce stade, le seul argument valable pour préférer la solution précédente est le fait qu'en principe on construit moins souvent qu'on n'utilise.

Pour les tableaux, ça se corse plus :

⟨*bidon*⟩+≡
```
typedef struct ap_ap_lincons1_array_t {
  ap_lincons1_t* array;
  size_t size;
};
```

Pour convertir du niveau 1 au niveau 0, cela nécessite une allocation mémoire, et tous les ennuis pour désallouer après utilisation.

⟨*bidon*⟩+≡
```
ap_ap_lincons0_array_t ap_ap_lincons0_array_of_ap_lincons1_array(lincons_array1_t* array1){
  ap_ap_lincons0_array_t array0 = ap_ap_lincons0_array_make(array1->size);
  /* Remplir, vérifier que les environnnemnts sont égaux, etc */
}
```

Retourner la contrainte de niveau 1 correspondant à l'index i d'un tableau est immédiat. L'affectation de l'index i est immédiat aussi, mais on ne vérifie pas que les environnements sont les mêmes.

Chapitre 10

Interface du domaine abstrait, niveau 1 : fichier `ap_abstract1.h`

L'interface de niveau 1 à une librairie sous-jacente est entièrement générique : elle se contente de traduire les opérations sur des objets de niveau 1 en objets de niveau 0.

⟨*⟩≡

```
/* ********************************************************************** */
/* ap_abstract1.h : generic operations on abstract values at level 1 */
/* ********************************************************************** */

/* This file is part of the APRON Library, released under LGPL license.  Please
   read the COPYING file packaged in the distribution */

/* GENERATED FROM ap_abstract1.nw : DO NOT MODIFY! */

#ifndef _AP_ABSTRACT1_H_
#define _AP_ABSTRACT1_H_

#include "ap_manager.h"
#include "ap_abstract0.h"
#include "ap_expr1.h"

#ifdef __cplusplus
extern "C" {
#endif
```

10.1 Type de donnée et généralités

66. **Valeur abstraite de niveau 1.**

On procède comme dans le module `expr1` : on rajoute un environnement, qui permet d'effectuer la liaison entre noms et dimensions.

⟨*⟩+≡
```
typedef struct ap_abstract1_t {
  ap_abstract0_t* abstract0;
  ap_environment_t* env;
} ap_abstract1_t;
  /* data structure invariant :
     ap_abstract0_integer_dimension(man,abstract0)== env->intdim &&
     ap_abstract0_real_dimension(man,abstract0)== env->realdim */

typedef struct ap_box1_t {
  ap_interval_t** p;
  ap_environment_t* env;
} ap_box1_t;
void ap_box1_fprint(FILE* stream, const ap_box1_t* box);
void ap_box1_clear(ap_box1_t* box);
```

67. **Principes suivis.**

Les principales fonctionnalités du niveau 1 sont :
- D'effectuer la conversion des noms vers les dimensions,
- De redimensionner les valeurs abstraites et les expressions, contraintes, ... définies avec des environnements différents, de telle sorte qu'elles soient définies sur un environnement commun (le plus petit environnement commun, s'il existe ; dans le cas contraire, une exception `INVALID_ARGUMENT` est lancée).

La politique de redimensionnement choisie est la suivante :
- Pour les fonctions prenant en argument une valeur abstraite et une expression (ou contrainte, ou générateur, ou tableau de ...), on suppose que l'environnement de l'expression est un sous-environnement de la valeur abstraite (sinon exception). Si c'est un sous-environnement strict, on redimensionne l'expression.

  Motivation : d'une part la simplicité, d'autre part le peu d'utilité pratique (c'est BJ qui parle). Exemples avec une politique plus laxiste (sur des polyèdres définis sur x et y) :

  1. ap_abstract1_meet_lincons1($\{1 \leq x \leq y \leq 3\}, z \geq y$) donnerait une nouvele valeur abstraite avec z non contraint.
  2. ap_abstract1_assign_linexpr($\{1 \leq x \leq y \leq 3\}, y, x + z$) donnerait une nouvele valeur abstraite avec y et z non contraints.

  Il faut remarquer qu'on peut traiter ces cas à la main, en redimensionant explicitement les polyèdres.
- Pour les fonctions prenant en argument plusieurs valeurs abstraites, on exige qu'elles soient toutes définies sur le même environnement.

Du point de vue de l'efficacité, le redimmensionnement a un certain coût (calculs sur les environnements, allocation et déallocation d'objets temporaires). C'est de la responsabilité de l'utilisateur soit d'être précis sur les environnements et de ne pas en changer tout le temps (efficacité), soit d'être moins attentif au prix d'un certain coût.

10.2 Gestion mémoire, Représentations, Entrées/Sorties

⟨*⟩+≡

```
/* ********************************************************************** */
/* I. General management */
/* ********************************************************************** */
```

68. Gestion mémoire

⟨*⟩+≡

```
/* ============================================================ */
/* I.1 Memory */
/* ============================================================ */

ap_abstract1_t ap_abstract1_copy(ap_manager_t* man, const ap_abstract1_t* a);
  /* Return a copy of an abstract value, on
     which destructive update does not affect the initial value. */

void ap_abstract1_clear(ap_manager_t* man, ap_abstract1_t* a);
  /* Free all the memory used by the abstract value */

size_t ap_abstract1_size(ap_manager_t* man, const ap_abstract1_t* a);
  /* Return the abstract size of an abstract value (see ap_manager_t) */
```

69. Control of internal representation

⟨*⟩+≡

```
/* ============================================================ */
/* I.2 Control of internal representation */
/* ============================================================ */

void ap_abstract1_minimize(ap_manager_t* man, const ap_abstract1_t* a);
  /* Minimize the size of the representation of a.
     This may result in a later recomputation of internal information.
  */

void ap_abstract1_canonicalize(ap_manager_t* man, const ap_abstract1_t* a);
  /* Put the abstract value in canonical form. (not yet clear definition) */

void ap_abstract1_approximate(ap_manager_t* man, ap_abstract1_t* a, int algorithm);
  /* Perform some transformation on the abstract value, guided by the
     field algorithm.

     The transformation may lose information.  The argument "algorithm"
     overrides the field algorithm of the structure of type foption_t
     associated to ap_abstract1_approximate (commodity feature). */

tbool_t ap_abstract1_is_minimal(ap_manager_t* man, const ap_abstract1_t* a);
tbool_t ap_abstract1_is_canonical(ap_manager_t* man, const ap_abstract1_t* a);
```

Les propriétés de la forme canonique ne sont pas encore bien claires (*c.f.* §1.4). On aimerait que par exemple la sérialisation en binaire d'un objet canonique soit aussi canonique.

70. **Impression.**

⟨*⟩+≡

```
/* ============================================================ */
/* I.3 Printing */
/* ============================================================ */

void ap_abstract1_fprint(FILE* stream,
                         ap_manager_t* man,
                         const ap_abstract1_t* a);
  /* Print the abstract value in a pretty way */

void ap_abstract1_fprintdiff(FILE* stream,
                             ap_manager_t* man,
                             const ap_abstract1_t* a1, const ap_abstract1_t* a2);
  /* Print the difference between a1 (old value) and a2 (new value).
     The meaning of difference is library dependent. */

void ap_abstract1_fdump(FILE* stream, ap_manager_t* man, const ap_abstract1_t* a);
  /* Dump the internal representation of an abstract value,
     for debugging purposes. */
```

71. **Précisions sur les fonctions d'impressions.** Le format des fonctions d'impression est propre à chaque librairie : aucune syntaxe n'est imposée, ne serait-ce que parce que la représentation interne des nombres n'est pas unifiée.

Si l'on désire une syntaxe uniforme, il faut d'abord convertir en contraintes/générateurs utilisateur.

??

72. **Sérialisation**

La sérialisation/désérialisation en binaire, comme déjà mentionné, ne fonctionne que pour la même librairie, compilée avec les mêmes options de représentation interne des nombres.

Question : doit-on sérialiser l'environnement associé ? (BJ : oui)

⟨*⟩+≡

```
/* ============================================================ */
/* I.4 Serialization */
/* ============================================================ */

ap_membuf_t ap_abstract1_serialize_raw(ap_manager_t* man, const ap_abstract1_t* a);
/* Allocate a memory buffer (with malloc), output the abstract value in raw
   binary format to it and return a pointer on the memory buffer and the size
   of bytes written.  It is the user responsability to free the memory
   afterwards (with free). */

                                                                    ap_abstract1_t ap_abst
/* Return the abstract value read in raw binary format from the input stream
   and store in size the number of bytes read */
```

10.3 Constructeurs, accesseurs, tests et extraction de propriétés

73. **Basic constructors**

⟨*⟩+≡

```
/* ********************************************************************** */
/* II. Constructor, accessors, tests and property extraction */
/* ********************************************************************** */

/* ============================================================ */
/* II.1 Basic constructors */
/* ============================================================ */

ap_abstract1_t ap_abstract1_bottom(ap_manager_t* man, ap_environment_t* env);
  /* Create a bottom (empty) value defined on the environment */

ap_abstract1_t ap_abstract1_top(ap_manager_t* man, ap_environment_t* env);
  /* Create a top (universe) value defined on the environment */

ap_abstract1_t ap_abstract1_of_box(ap_manager_t* man,
                                   ap_environment_t* env,
                                   ap_var_t* tvar,
                                   ap_interval_t** tinterval,
                                   size_t size);
  /* Abstract an hypercube defined by the arrays tvar and tinterval,
     satisfying : forall i, tvar[i] in tinterval[i].

     If no inclusion is specified for a variable in the environement, its value
     is no constrained in the resulting abstract value.
  */

ap_abstract1_t ap_abstract1_of_lincons_array(ap_manager_t* man,
                                             ap_environment_t* env,
                                             const ap_lincons1_array_t* array);
  /* Abstract a convex polyhedra defined on the environment
     by the array of linear constraints
  */
```

74. **Accessors**

⟨*⟩+≡

```
/* ============================================================ */
/* II.2 Accessors */
/* ============================================================ */

const ap_environment_t* ap_abstract1_environment(ap_manager_t* man, const ap_abstract1_t* a);
const ap_abstract0_t* ap_abstract1_abstract0(ap_manager_t* man, const ap_abstract1_t* a);
```

75. **Tests**

⟨*⟩+≡

```
/* ========================================================== */
/* II.3 Tests */
/* ========================================================== */

/* If any of the following functions returns tbool_top, this means that
   an exception has occured, or that the exact computation was
   considered too expensive to be performed (according to the options).
   The flag exact and best should be cleared in such a case. */

tbool_t ap_abstract1_is_bottom(ap_manager_t* man, const ap_abstract1_t* a);
tbool_t ap_abstract1_is_top(ap_manager_t* man, const ap_abstract1_t* a);

tbool_t ap_abstract1_is_leq(ap_manager_t* man, const ap_abstract1_t* a1, const ap_abstract1_t* a2);
  /* inclusion check */

tbool_t ap_abstract1_is_eq(ap_manager_t* man, const ap_abstract1_t* a1, const ap_abstract1_t* a2);
  /* equality check */

tbool_t ap_abstract1_sat_lincons(ap_manager_t* man, const ap_abstract1_t* a, const ap_lincons1_t* lin
  /* does the abstract value satisfy the linear constraint? */

tbool_t ap_abstract1_sat_interval(ap_manager_t* man, const ap_abstract1_t* a,
                                  ap_var_t var, const ap_interval_t* interval);
  /* is the dimension included in the interval in the abstract value?
     - Raises an exception if var is unknown in the environment of the abstract value
  */
tbool_t ap_abstract1_is_variable_unconstrained(ap_manager_t* man, const ap_abstract1_t* a,
                                               ap_var_t var);
  /* is the variable unconstrained in the abstract value?
     - Raises an exception if var is unknown in the environment of the abstract value
  */
```

76. **Extraction de propriétés**

⟨*⟩+≡

```
/* ============================================================ */
/* II.4 Extraction of properties */
/* ============================================================ */

ap_interval_t* ap_abstract1_bound_linexpr(ap_manager_t* man,
                                    const ap_abstract1_t* a, const ap_linexpr1_t* expr);
  /* Returns the interval taken by a linear expression
     over the abstract value */

ap_interval_t* ap_abstract1_bound_variable(ap_manager_t* man,
                                           const ap_abstract1_t* a, ap_var_t var);
  /* Returns the interval taken by the variable
     over the abstract value
     - Raises an exception if var is unknown in the environment of the abstract value
  */

ap_lincons1_array_t ap_abstract1_to_lincons_array(ap_manager_t* man, const ap_abstract1_t* a);
  /* Converts an abstract value to a polyhedra
     (conjunction of linear constraints).
     - The environment of the result is a copy of the environment of the abstract value.
  */

ap_box1_t ap_abstract1_to_box(ap_manager_t* man, const ap_abstract1_t* a);
  /* Converts an abstract value to an interval/hypercube. */

ap_generator1_array_t ap_abstract1_to_generator_array(ap_manager_t* man, const ap_abstract1_t* a);
  /* Converts an abstract value to a system of generators.
     - The environment of the result is a copy of the environment of the abstract value.
 */
```

10.4 Opérations

77. Bornes supérieures et inférieures

⟨*⟩+≡

```
/* ********************************************************************** */
/* III. Operations : functional version */
/* ********************************************************************** */

/* ============================================================ */
/* III.1 Meet and Join */
/* ============================================================ */

ap_abstract1_t ap_abstract1_meet(ap_manager_t* man, bool destructive, ap_abstract1_t* a1, const ap_abs
ap_abstract1_t ap_abstract1_join(ap_manager_t* man, bool destructive, ap_abstract1_t* a1, const ap_abs
  /* Meet and Join of 2 abstract values
     - The environment of the result is the lce of the arguments
     - Raises an EXC_INVALID_ARGUMENT exception if the lce does not exists
  */

ap_abstract1_t ap_abstract1_meet_array(ap_manager_t* man, const ap_abstract1_t* tab, size_t size);
ap_abstract1_t ap_abstract1_join_array(ap_manager_t* man, const ap_abstract1_t* tab, size_t size);
  /* Meet and Join of an array of abstract values.
     - Raises an [[exc_invalid_argument]] exception if [[size==0]]
       (no way to define the dimensionality of the result in such a case
     - The environment of the result is the lce of the arguments
     - Raises an EXC_INVALID_ARGUMENT exception if the lce does not exists
  */

ap_abstract1_t ap_abstract1_meet_lincons_array(ap_manager_t* man,
                                               bool destructive,
                                               ap_abstract1_t* a,
                                               const ap_lincons1_array_t* array);
  /* Meet of an abstract value with a set of constraints
     (generalize ap_abstract1_of_lincons_array) */

ap_abstract1_t ap_abstract1_add_ray_array(ap_manager_t* man,
                                          bool destructive,
                                          ap_abstract1_t* a,
                                          const ap_generator1_array_t* array);
  /* Generalized time elapse operator */
```

78. **Affectations et Substitutions.**

⟨*⟩+≡

```
/* ========================================================== */
/* III.2 Assignement and Substitutions */
/* ========================================================== */

/* Assignement and Substitution of a single
   dimension by a interval linear expression */
ap_abstract1_t ap_abstract1_assign_linexpr(ap_manager_t* man,
                                  bool destructive, ap_abstract1_t* a,
                                  ap_var_t var, const ap_linexpr1_t* expr,
                                  const ap_abstract1_t* dest);
ap_abstract1_t ap_abstract1_substitute_linexpr(ap_manager_t* man,
                                      bool destructive, ap_abstract1_t* a,
                                      ap_var_t var, const ap_linexpr1_t* expr,
                                      const ap_abstract1_t* dest);

/* Parallel Assignement and Substitution of several dimensions by
   linear expressons. */
ap_abstract1_t ap_abstract1_assign_linexpr_array(ap_manager_t* man,
                                        bool destructive, ap_abstract1_t* a,
                                        ap_var_t* tvar,
                                        const ap_linexpr1_t* texpr,
                                        size_t size,
                                        const ap_abstract1_t* dest);
ap_abstract1_t ap_abstract1_substitute_linexpr_array(ap_manager_t* man,
                                            bool destructive, ap_abstract1_t* a,
                                            ap_var_t* tvar,
                                            const ap_linexpr1_t* texpr,
                                            size_t size,
                                            const ap_abstract1_t* dest);
```

79. **Projections et Quantifications existentielles.**

⟨*⟩+≡

```
/* ========================================================== */
/* III.3 Projections */
/* ========================================================== */

ap_abstract1_t ap_abstract1_forget_array(ap_manager_t* man,
                                  bool destructive, ap_abstract1_t* a,
                                  ap_var_t* tvar, size_t size,
                                  bool project);
```

80. Changement d'environnement

⟨*⟩+≡

```
/* ============================================================ */
/* III.4 Change of environnement */
/* ============================================================ */

ap_abstract1_t ap_abstract1_change_environment(ap_manager_t* man,
                                               bool destructive, ap_abstract1_t* a,
                                               ap_environment_t* nenv,
                                               bool project);

ap_abstract1_t ap_abstract1_minimize_environment(ap_manager_t* man,
                                                 bool destructive, ap_abstract1_t* a);
  /* Remove from the environment of the abstract value
     variables that are unconstrained in it. */

ap_abstract1_t ap_abstract1_rename_array(ap_manager_t* man,
                                        bool destructive, ap_abstract1_t* a,
                                        ap_var_t* var, ap_var_t* nvar, size_t size);
  /* Parallel renaming. The new variables should not interfere with the variables that are not renamed
```

81. Expansion et pliage

⟨*⟩+≡

```
/* ============================================================ */
/* III.5 Expansion and folding of dimensions */
/* ============================================================ */

ap_abstract1_t ap_abstract1_expand(ap_manager_t* man,
                                   bool destructive, ap_abstract1_t* a,
                                   ap_var_t var,
                                   ap_var_t* tvar, size_t size);
  /* Expand the variable var into itself + the size additional variables
     of the array tvar, which are given the same type as var.

     It results in (size+1) unrelated variables having same
     relations with other variables.

     The additional variables are added to the environment
     of the argument for making the environment of the result.
  */

ap_abstract1_t ap_abstract1_fold(ap_manager_t* man,
                                 bool destructive, ap_abstract1_t* a,
                                 ap_var_t* tvar, size_t size);
  /* Fold the dimensions in the array tvar of size n>=1 and put the result
     in the first variable in the array.

     The other variables of the array are then forgot
     and removed from the environment. */
```

82. **Élargissement.**

⟨*⟩+≡

```
/* ========================================================== */
/* III.6 Widening */
/* ========================================================== */

/* Widening */
ap_abstract1_t ap_abstract1_widening(ap_manager_t* man,
                                const ap_abstract1_t* a1, const ap_abstract1_t* a2);
/* Widening with threshold */
ap_abstract1_t ap_abstract1_widening_threshold(ap_manager_t* man,
                                const ap_abstract1_t* a1, const ap_abstract1_t* a2,
                                const ap_lincons1_array_t* array);
```

83. **Clôture topologique.**

⟨*⟩+≡

```
/* ========================================================== */
/* III.7 Closure operation */
/* ========================================================== */

/* Returns the topological closure of a possibly opened abstract value */

ap_abstract1_t ap_abstract1_closure(ap_manager_t* man, bool destructive,  ap_abstract1_t* a);
```

⟨*⟩+≡

```
#ifdef __cplusplus
}
#endif

#endif
```

Chapitre 11

Implementor manual

11.1 How to make an existing library conformant ?

We briefly describe here how to connect an existing library to the common interface.

First, the library has to expose an interface which conforms to the level 0 of the interface (module `abstract0`). All the functions described in this module should be defined. If a function is not really implemented, at least it shoulld contain the code raising the exception `EXC_NOT_IMPLEMENTED`. The implementor may use any functions of the modules `coeff`, `expr0` and `manager` to help the job of converting datatypes of the interface to internal datatypes used inside the library.

Second and last, the library should expose an initialization function that allocates and initializes properly an object of type `manager_t`. For this purpose, the module `manager` offers the utility functions `manager_alloc`. As an example, we give the definition of the function allocating a manager as implemented in the NewPolka.

1. Header of the function :
   ⟨*init1*⟩≡
   ```
   manager_t* pk_manager_alloc(
     bool strict /* specific parameter : do we allow strict constaints? */
   )
   {
   ```

2. Allocation et initialisation des données globales spécifiques à NewPolka :
   ⟨*init1*⟩+≡
   ```
   pk_internal_t* pk = pk_internal_alloc(strict); /* allocation */
   pk_set_approximate_max_coeff_size(pk, 1);
     /* initialization of specific functions
        (not offered in the common interface) */
   ```

3. Allocation of the manager itself :
   ⟨*init1*⟩+≡
   ```
   manager_t* man = manager_alloc("polka","2.0",
                                  pk, (void (*)(void*))pk_internal_free);
   ```

We provide resp. name, version, internal specific manager, and the function to free it.

The function `manager_alloc` sets the options of the commoninterface to their default value (see documentation).

4. Initialization of the "virtual" table : we need to connect the generic functions of the interface (eg, `abstract_meet`, ...) to the actual functions of the library.

⟨*init1*⟩+≡
```
    funptr = man->funptr;

    funptr[fun_minimize] = &poly_minimize;
    funptr[fun_canonicalize] = &poly_canonicalize;
    funptr[fun_approximate] = &poly_approximate;
    funptr[fun_is_minimal] = &poly_is_minimal;
    funptr[fun_is_canonical] = &poly_is_canonical;
    funptr[fun_fprint] = &poly_fprint;
    funptr[fun_fprintdiff] = &poly_fprintdiff;
    funptr[fun_fdump] = &poly_fdump;
    ...
```

5. Last, we return the allocated manager :

⟨*init1*⟩+≡
```
    return man;
  }
```

That's all for the implementor side.

11.2 User side : how to use the common interface

From the user point of view, the benefit of using the common interface is to restrict the place where the user is aware of the real library in use is located in the allocation and initialization of the manager.

⟨*use1*⟩≡
```
/* Allocating a manager using Polka functions */
manager_t* man = pk_manager_alloc(true);
/* Setting options offered by the common interface,
   but with meaning possibly specific to the library */
manager_set_abort_if_exception(man,EXC_OVERFLOW,true);
{
  funopt_t funopt;
  funopt_init(&funopt);
  funopt.algorithm = 1; /* default value is 0 */
  manager_set_funopt(man,fun_widening,&funopt); /* Setting options for widening */
}
{
  funopt_t funopt = manager_get_funopt(man,fun_widening);
  funopt.timeout = 30;
  manager_set_funopt(man,fun_widening,&funopt);
}
/* Obtaining the internal part of the manager and setting specific options */
pk_internal_t* pk = manager_get_internal(man);
pk_set_max_coeff_size(pk,size);
```

The standard operations can then be used and will have the semantics defined in the interface. Notice however that some generic functions are not formally generic : `abstract_fprint`, `abstract_fdump`, `abstract_approximate`.

At any point, options may be modified in the same way as during the initialization.

84. **Typing issue.**

The use of several libraries at the same time via the common interface and the managers associated to each library raises the problem of typing. Look at the following code :

⟨*type error*⟩≡
```
manager_t* manpk = pk_manager_alloc(true); /* manager for Polka */
manager_t* manoct = oct_manager_alloc();    /* manager for octagon */

abstract0_t* abs1 = abstract_top(manpk,3,3);
abstract0_t* abs2 = abstract_top(manoct,3,3);
tbool_t tb = abstract0_is_eq(manoct,abs1,abs2);
  /* Problem : the effective function called (octagon_is_eq) expects
     abs1 to be an octagon, and not a polyhedron! */

abstract0_t* abs3 = abstract_top(manoct,3,3);
abstract0_meet_with(manpk,abs2,abs3);
  /* Problem : the effective function called (pk_meet_with) expects
     abs2 and abs3 to be polyhedra, but they are octagons */
```

There is actually no typing, as `abstract0_t*`, `abstract1_t` and `manager_t` are abstract types shared by the different libraries. Notice that the use of C++ and inheritance would not solve directly the problem, if functions of the interface are methods of the manager ; one would have :

⟨*type error2*⟩≡
```
manager_t* manpk = pk_manager_alloc(true);
  /* manager for Polka, effective type pk_manager_t* */
manager_t* manoct = oct_manager_alloc();
  /* manager for octagon, effective type oct_manager_t* */

abstract0_t* abs1 = manpk->abstract_top(3,3);
  /* effective type : poly_t */
abstract0_t* abs2 = manoct->abstract_top(3,3);
  /* effective type : oct_t */
tbool_t tb = manoct->abstract0_is_eq(abs1,abs2);
  /* No static typing possible :
     manpk->abstract0_is_eq and manoct->abstract0_is_eq should have the same
     signature (otherwise one cannot interchange manpk and manoct in the code),
     which means that abs1 and abs2 are supposed to be of type abstract0_t* */
*/
```

85. **Choices made wrt typing.**

The C interface does not perform any typing check. If one would wish it, a simple solution would be the one adopted for the OCaml interface (see below).

The OCaml interface does perform a *dynamic* typing check. An abstract value of type `Abstract0.t` contain both a C pointer to an object of type `abstract0_t` and a C pointer to an object of type `manager_t`.

1. This was anyway needed to finalize `Abstract0.t` object, because one need to access to the effective finalization function, which is done here by calling `(man->funptr[fun_free])(man,abs)`.
2. A benefit is that one do not need to always pass explicitly the manager as an argument to the functions. We just have

   ⟨*<example>*⟩≡
   ```
   (* Module Abstract0 *)
   type t

   val top : Manager.t -> t
     (* Here we need to specify the manager *)
   val meet : t -> t -> t
     (* Here we just have to check the compatibility of the managers
        contained in the arguments *)
   ```
3. The compatibility of the manager adopted is the following one : `man1->funptr==man2->funptr` : the virtual table should be the same. Checking that `man1==man2` would be too strong, because one may use two managers created by the same library, to avoid interference in multithreaded programming.